Practical Semantic Web and Linked Data Applications

Common Lisp Edition

Uses the Free Editions of Franz Common Lisp and AllegroGraph

Mark Watson

November 3, 2010

Contents

List of Figures

List of Tables

Preface

This book is primarily intended to be a practical guide for using RDF data in information processing, linked data, and semantic web applications using the Common Lisp APIs for the AllegroGraph product. A second use for this book is to help you, the reader, set up an interactive Lisp development environment for writing knowledge intensive applications. So, while the Semantic Web applications using the AllegroGraph RDF data store is the main theme in this book, I will also cover using data from a variety of sources like Freebase, DBpedia and other public RDF repositories, use of statistical Natural Language Processing (NLP), and the GeoNames database and public web service.[1]

1. Getting started

I expect you to install the Franz Free Edition Common Lisp environment with the Free Edition of AllegroGraph to work through the examples in this book. If you own professional or enterprise licenses for these projects you can use that also. The free editions have restrictions on their use so read the license agreement.

Download the Free Lisp Edition and then install AllegroGraph using:

```
(require :update)
(system.update:install-allegrograph)
```

Whenever you start Lisp using evaluate the following form to load AllegroGraph:

```
(require :agraph)
```

I do not duplicate information in this book that appears in the documentation available on the Franz web site. I urge you to read how to set up an Emacs development environment.

[1] The geonames.org web service is limited to 2000 queries per hour from any single IP address. Commercial support is available, or, with some effort, you can also run GeoNames on your own server.

2. Portable Common Lisp Code Book Examples

Even though many examples in this book use AllegroGraph I also provide many portable Common Lisp examples and utilities that I have written for my own work and research:

1. KnowledgeBooks.com Lisp Natural Language Processing (NLP) library
2. Client library for using the Open Calais web service[2][3]
3. Example code for using the Freebase web service[4]
4. Example code for using DBpedia web services[5]
5. Example code for using GeoBase.org web services[6]

3. Using the Common Lisp ASDF Package Manager

There are several package managers available for Common Lisp and I have chosen to use ASDF for the examples in this book that are not self contained in a single source directory. ASDF uses a search path list as a source of directories to find package definition files (that end with the extension *.asd*). For instance, since some of the examples in this book will need to make web service calls I have an example directory *aserve_client* to show you how to use Franz's open source **aserve** client library. The example code needs to use the Yason JSON parsing and generation library in *utils/yson*:

```
(push "../utils/yason/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op 'yason)
```

The first statement pushes the Yson package directory on the ASDF search path list and the second line loads the package named *yson*.

[2] Requires the open source Portable AllegroServe and split-sequence libraries.

[3] The example program that puts entities found in text into an RDF data store requires the AllegroGraph library.

[4] Requires the Franz Portable AllegroServe client library. This is open source, but you will need to manually install Portable AllegroServe if you are using an alternative Common Lisp implementation (e.g., SBCL or Clozure Common Lisp).

[5] Requires the AllegroGraph SPARQL client APIs but this code could be rewritten.

[6] Requires several open source libraries that are included in the ZIP file for this book's examples: cl-json, s-xml, split-sequence, usocket, trivial-gray-streams, flexi-streams, chunga, cl-base64, puri, drakma, and cl-geonames.

When you browse through the directories containing the examples in this book you will notice that I use the convention of placing short snippets of test code that I often include in the book text in files named *test.lisp* and put longer examples in files named *example.lisp*.

This book is organized in layers:

1. Quick introduction to the AllegroGraph and Franz Lisp.
2. Theory (with some AllegroGraph-specific short examples).
3. Detailed treatment of AllegroGraph APIs.
4. Development of Useful Common Lisp libraries information processing, and importing linked data sources like Freebase and Open Calais data to an AllegroGraph RDF store.
5. Development of a complete web portal using Semantic Web technologies.

4. Information on the Companion Edition to this Book that Covers Java and JVM Languages

This book has a companion edition that covers the use of both AllegroGraph and the open source Sesame project using JVM based languages like Java, Clojure, JRuby, and Scala. If you primarily work with JVM languages then you will likely be better off working through the other edition of this book. The JVM edition of this book offers some portability: I provide the basic functionality of AllegroGraph for RDF storage, SPARL queries, Geo Location, and free text search using the Sesame RDF data store and my own search and Geo Location libraries.

The Free Java Edition of AllegroGraph does not have the commercial use restrictions that the Free Lisp Edition does.

If you want a free commercial friendly Lisp development and deployment environment: I recommend that you use the companion book with the Clojure programming language.

5. AllegroGraph

AllegroGraph is written in Common Lisp and comes bundled in several different products:

1. As a standalone server that supports Lisp, Ruby, Java, Clojure, Scala, Common Lisp, and Python clients. A free version (limited to 50 million RDF triples - a large limit) that can be used for any purpose, including commercial use.

2. The WebView interface for exploring, querying, and managing AllegroGraph triple stores. WebView is standalone because it contains an embedded AllegroGraph server.

3. The Gruff for exploring, querying, and managing AllegroGraph triple stores using table and graph views. Gruff is standalone because it contains an embedded AllegroGraph server.

4. A library that is used embedded in Franz Common Lisp applications. A free version is available (with some limitations) for non-commercial use.

This book uses AllegroGraph in embedded mode because this is the most agile way to experiment and learn the APIs. What you learn in this book is applicable to all of the AllegroGraph products. Currently, the server release is version 4.0 and the embedded library release is 3.3.

6. Software License for Example Code in this Book

The small example code snippets listed in this book text and my code for the larger example applications and libraries in the code ZIP file are licensed using the AGPL.[7]

Commercial waiver of some AGPL terms for people or organizations who purchase this book:

If you purchase the print edition or purchase the PDF file[8] of this book then I grant you a partial commercial use waiver to the AGPL deploying your applications on a single server: you can use my examples in applications on a single server without the requirement of releasing the source code for your application under the AGPL (all other AGPL license terms apply). If you need to run an application using my code on multiple servers, then please purchase one copy of the book for each server.

I enjoy writing and purchasing copies of this book helps fund future writing projects.

Acknowledgements

I would like to thank my wife Carol Watson for copyediting this book.

[7] For your convenience, I include in the code ZIP file third party libraries, most of which are released under MIT, BSD, Lisp LGPL, or Apache licenses.

[8] downloading the free PDF from http://markwatson.com/opencontent does not give you the rights to this waiver.

1. Introduction

Franz has good online documentation[1] for all of their AllegroGraph products. One purpose of this book is to provide a brief introduction to AllegroGraph but I assume that you also reference the documentation on the Franz web site. The broader purpose of this book is to provide application programming examples using AllegroGraph and Linked Data sources on the web. This book also covers some of my own open source Common Lisp projects that you may find useful for Semantic Web applications. The combination of interactive Lisp development with embedded AllegroGraph and my utilities covered later should provide you with an agile development environment for writing knowledge based and semantic web applications.

AllegroGraph is an RDF data repository that can use RDFS and RDFS+ inferencing. AllegroGraph also provides three non-standard extensions:

1. Text indexing and search
2. Geo Location support
3. Network traversal and search for social network applications

1.1. Who is this Book Written For?

I assume that you both already know how to program in Common Lisp and that you write applications that require handling large amounts of unstructured information. AllegroGraph is a powerful tool for handling large amounts of data and Lisp programming environments are excellent for rapidly prototyping new applications. Along with extra libraries I have written for using linked data sources on the web, this book will hopefully provide you with new tools to rapidly solve application problems that would be more difficult to handle using relational databases.

Franz also provides support for embedding AllegroGraph in Lisp applications and for using it in a client mode with external AllegroGraph servers. Since the APIs are almost identical, I take a shortcut in writing this book and concentrate on using AllegroGraph in embedded mode.

[1] http://franz.com/agraph/support/documentation/current/agraph-introduction.html

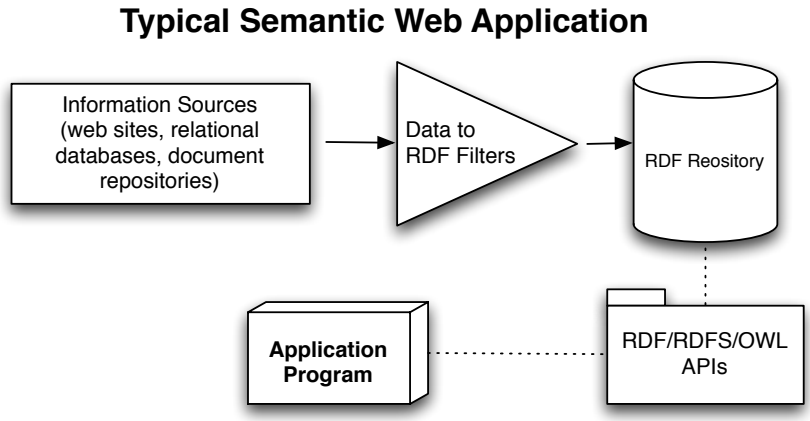

Figure 1.1.: Example Semantic Web Application

There are many books, good tutorials and software about the Semantic Web on the web. However, there is not a single reference for developers who want to use the combination of Common Lisp and AllegroGraph for development using technologies like RDF/RDFS/OWL modeling, descriptive logic reasoners, and the SPARQL query language.

If you own a Franz Lisp and AllegroGraph development license, then you are set to go. If not, you need to download and install a free edition copy at:

http://www.franz.com/downloads/

You may also want to download and install the free versions of the AllegroGraph standalone server, Gruff, and WebView.[2]

Franz Inc. has provided support for my writing this book in the form of technical reviews and my understanding is that even though you will need to periodically refresh your free non-commercial license, there is no inherent time limit for non-commercial use. I would also like to thank Franz for providing me with an Enterprise developers license for my MacBook that I use for my own research and development projects.

[2] I do not use these associated products in this book but I do in the Java, Clojure, Scala, and JRuby edition of this book.

1.2. Why a PDF Copy of this Book is Available Free on the Web

As an author I want to earn a living writing and have many people read and enjoy my books. By offering for sale the print version of this book I can earn some money for my efforts and also allow readers who can not afford to buy many books or may only be interested in a few chapters to read it from my web site. If you support my future writing projects by purchasing either the print or PDF version of this book, I thank you by offering you more flexibility in the software license terms for the example programs and libraries I developed (see Section 6 in the Preface).

Please note that I do not give permission to post the PDF version of this book on other people's web sites: I consider this to be at least indirectly commercial exploitation in violation the Creative Commons License that I have chosen for this book.

1.3. Book Software

You can download a large ZIP file containing all code and test data from the URL:

```
http://markwatson.com/opencontent/lisp_semantic_web_code.zip
```

The book example code, libraries, and applications are organized in subdirectories organized by topic:

1. dbpedia - use the DBPedia web services
2. freebase_client - use the Freebase web services
3. geonames - use the Geonames web service
4. knowledgebooks_nlp - my natural language processing library
5. opencalais - use the OpenCalais web services
6. quick_start_allegrograph_lisp_embedded - code snippets used to introduce Allegrograph
7. quick_start_allegrograph_standalone_server - code snippets for Chapter 2
8. rdf - additional code snippets for created RDF triples and making queries
9. reasoning - code snippets for Chapter 8
10. sparql - code snippets and sample data for SPARQL queries

11. test_data - miscellaneous test data files
12. utils - third party libraries[3] that I use for the book examples
13. web_app - both backend code from Chapter 16 and the front end web application code from Chapter 17

1.4. Why Graph Data Representations are Better than the Relational Database Model for Dealing with Rapidly Changing Data Requirements

When people are first introduced to Semantic Web technologies their first reaction is often something like, "I can just do that with a database." The relational database model is an efficient way to express and work with slowly changing data models. There are some clever tools for dealing with data change requirements in the database world (ActiveRecord and migrations being a good example) but it is awkward to have end users and even developers tagging on new data attributes to relational database tables.

A major theme in this book is convincing you that modeling data with RDF and RDFS facilitates freely extending data models and also allows fairly easy integration of data from different sources using different schemas without explicitly converting data from one schema to another for reuse. You will learn how to use the SPARQL query language to use information in different RDF repositories. It is also possible to publish relational data with a SPARQL interface. [4]

1.5. What if You Use Other Programming Languages Other Than Lisp?

If you are a Java programmer, you probably still want to learn about AllegroGraph because Franz distributes a free Java version of AllegroCache that can be used for any purposes (including commercial applications) – the free Java version is limited to 50 million RDF triples. The Java version is a natively compiled Franz Lisp application that provides plain socket and HTTP/REST interfaces.

[3] cl-json, s-xml, split-sequence, usocket, trivial-gray-streams, flexi-streams, chunga, cl-base64, puri, drakma, and cl-geonames

[4] The open source D2R project provides a wrapper for relational databases that provides a SPARQL query interface.

If you do most of your development in other languages like Ruby and Python then you can run the free server edition using the HTTP/Sesame client protocol. Sesame is a high quality "batteries included" Java library for Semantic Web development; the Sesame client protocol is well documented and simple to use but will not be covered here. If you use the Sesame protocol then you have the flexibility of using both Franz's free server edition of AllegroGraph and Sesame which is open source with a BSD style license.

2. AllegroGraph Embedded Lisp Quick Start

The first section of this book will cover Semantic Web technologies from a theoretical and reference point of view. Since I want you to follow along with the book material as I present it, this chapter is intended to get you comfortable using Lisp and embedded AllegroGraph: it will be easier to work through the theory in Chapters 3, 4, and 6 if you understand the basics of AllegroGraph. After this more detailed look at some theory we will dig deeper into AllegroGraph development techniques in Chapters 7, 8, and 9.

2.1. Starting AllegroGraph

In this chapter and in much of this book, you can save some effort by copying and pasting the code snippets into the Lisp listener. The code snippets used in this chapter are contained in the source file **quick_start_lisp_embedded.lisp**. I assume that most readers are trying AllegroGraph using the free non-commercial use version so that is what I will use here. If you are using a commercially licensed version the examples will work the same but the initial banner display by *alisp* (conventional case insensitive Lisp shell) and *mlisp* ("modern" case sensitive Lisp shell) will be slightly different. While I usually use *alisp* in my work (I have been using Lisp for professional development since 1982), Franz recommends using *mlisp* for AllegroGraph development so we will use *mlisp* in this book. You will need to follow the directions in acl81_express/readme.txt to build a *mlisp* image to use. When showing interactive examples in this chapter I remove some Lisp shell messages so when you work along with these examples expect to see more output than what is shown here:[1]

```
markw$ mlisp
International Allegro CL Free Express Edition
8.2 [Mac OS X (Intel)] (Jul 9, 2009 17:15)
Copyright (C) 1985-2007, Franz Inc., Oakland, CA, USA.
                All Rights Reserved.
```

[1] I use OS X and Linux for my development. If you are a Windows user, follow the installation instructions on the AllegroGraph download web page and expect to see slight differences to the interactive example sessions that I use in this book.

```
This development copy of Allegro CL is licensed to:
   Trial User
;; Current reader case mode: :case-sensitive-lower
cl-user(1): (require :agraph)
AllegroGraph Lisp Edition 3.2 [built on March 16, 2009 15:05:15
t
cl-user(2): (in-package :db.agraph.user)
#<The db.agraph.user package>
TRIPLE-STORE-USER(3):
```

Please note that you will see many lines of output that I did not show. Here I required the **:agraph** package and changed the current Common Lisp package to **db.agraph.user**. In examples later in this book when we develop complete application examples we will be using our own application-specific packages and I will show you then what you need in general to import from $db.agraph$ and $db.agraph.user$. We will continue this interactive example Lisp session in the following sections.

I use interactive sessions in a command window for the examples in this book. If you are a Windows user then you will may want to alternatively try the Windows-specific IDE. I recommend that OS X, Linux, and Windows users use Emacs to develop Lisp code.[2]

If you run Franz Lisp in a terminal shell then I recommend that you start it using *rlwrap*. As an example, using OS X and Linux, I create an alias like:

```
alias lisp='rlwrap alisp'
```

Using *rlwrap* lets you use the up arrow key to rerun previous commands, edit previous commands, etc.

2.2. Working with RDF Data Stores

RDF data stores provide the services for storing RDF triple data and provide some means of making queries to identify some subset of the triples in the store. It is important to keep in mind that the mechanism for maintaining triple stores varies in different implementations. Triples can be stored in memory, in disk-based btree stores like BerkeleyDB, in relational databases, and in custom stores like AllegroGraph.

[2]Franz provides their own Emacs tools: look for instructions for installing ELI. However, I also use the SLIME Emacs Lisp development tools that are compatible with all versions of Lisp that I use: Franz, SBCL, ClozureCL, and Gambit-C Scheme. Franz provides SLIME installation instructions for Franz Common Lisp

While much of this book is specific to Common Lisp and AllegroGraph, the concepts that you will learn and experiment with can be useful if you also use other languages and platforms like Java (Sesame, Jena, OwlAPIs, etc.), Ruby (Redland RDF), etc. For Java developers Franz offers a Java version of AllegroGraph (implemented in Lisp with a network interface that also supports Python and Ruby clients) that I cover in the Java edition of this book.

2.2.1. Creating Repositories

AllegroGraph uses disk-based RDF storage with automatic in-memory caching. For the examples in this book I will assume that all RDF stores are kept in the temporary directory **/tmp**. For deployed systems you will clearly want to use a permanent location. For Windows(tm) development you can either change this location or create a new directory in **c:\tmp**. In the examples in this book, I assume a Mac OS X, Linux, or other Unix type file system:

```
TRIPLE-STORE-USER(3): (create-triple-store
                                "/tmp/rdfstore_1")
#<db.agraph::triple-db /tmp/rdfstore_1, open @ #x109682>
```

I hope that you are following along with this running example – you will better understand this material if you type it into a Lisp shell.

While it is possible to simultaneously work with multiple repositories (and this is well documented in Franz's online documentation for the non-free versions of AllegroGraph) for all of the tutorials, examples, and sample applications in this book we need just a single open repository in order to be compatible with the free versions of AllegroGraph.

We will see in Chapter 3 how to partition RDF triples into different namespaces and to use existing RDF data and schemas in different namespaces. In the following code snippet I introduce the AllegroGraph APIs for defining new namespaces and listing all namespaces defined in the current repository:

```
TRIPLE-STORE-USER(4): (register-namespace "kb"
                       "http://knowledgebooks.com/rdfs#")
"http://knowledgebooks.com/rdfs#"
TRIPLE-STORE-USER(5): (display-namespaces)
rdfs => http://www.w3.org/2000/01/rdf-schema#
err => http://www.w3.org/2005/xqt-errors#
fn => http://www.w3.org/2005/xpath-functions#
rdf => http://www.w3.org/1999/02/22-rdf-syntax-ns#
xs => http://www.w3.org/2001/XMLSchema#
```

```
xsd => http://www.w3.org/2001/XMLSchema#
owl => http://www.w3.org/2002/07/owl#
kb => http://knowledgebooks.com/rdfs#
```

Here I created a new name space that has an abbreviation (or nickname) **kb:** and then printed out all registered namespaces. To insure data integrity be sure to call **(close-triple-store)** to close an RDF triple store when you are done with it. I leave the connection open because we will continue to use it in this chapter.

2.2.2. AllegroGraph Lisp Reader Support for RDF

In general, the subject, predicate, and object parts of an RDF triple can be either URIs or literals.

AllegroGraph provides a Lisp reader macro **!** that makes it easier to enter URIs and literals. For example, the following two URIs are functionally equivalent given the **(register-namespace "kb" ...)** in the last section:

```
<http://knowledgebooks.com/rdfs#containsPerson>
!kb:containsPerson
```

String literals are also defined using the **!** reader macro; for example:

```
!"Barack Obama"
!"101 Main Street"
```

2.2.3. Adding Triples

A triple consists of a subject, predicate, and object. We refer to these three values as symbols **:s**, **:p**, and **:o** when using the AllegroGraph APIs. We saw the use of literals with the **!** Lisp reader macro in the last section. If we need to refer to either a subject, predicate, or object as a web URI then we use the function **resource**:

```
TRIPLE-STORE-USER(15): (resource "http://demo_news/12931")
!<http://demo_news/12931>
TRIPLE-STORE-USER(16): (defvar *demo-article*
                           (resource
                              "http://demo_news/12931"))
*demo-article*
TRIPLE-STORE-USER(17): *demo-article*
!<http://demo_news/12931>
```

The function **add-triple** takes three arguments for the subject, predicate, and object in a triple:

```
TRIPLE-STORE-USER(18): (add-triple *demo-article*
                        !rdf:type
                        !kb:article)
1
TRIPLE-STORE-USER(19): (add-triple *demo-article*
                         !kb:containsPerson
                         !"Barack Obama")
2
```

We used a combination of a generated resource, two predicates defined in the **rdf:** and **kb:** namespaces, and a string literal to define two triples. You notice that the function **add-triple** returns an integer as its value: this is a unique ID for the newly created triple.

2.2.4. Fetching Triples by ID

Triples in an AllegroGraph RDF store can be identified by a unique ID; this ID value is returned as the value of calling **add-triple** and can be used to fetch a triple:

```
TRIPLE-STORE-USER(20): (get-triple-by-id 2)
<12931 containsPerson Barack Obama>
TRIPLE-STORE-USER(21): (defvar *triple*
                         (get-triple-by-id 2))
*triple*
TRIPLE-STORE-USER(22): *triple*
<12931 containsPerson Barack Obama>
```

We will seldom access triples by ID – we will see shortly how to query a RDF store to find triples.

2.2.5. Printing Triples

The function **print-triple** can be used to print a short form of a triple and by adding the arguments **:format :concise** we can also print a triple in the NTriple format:

```
TRIPLE-STORE-USER(23): (print-triple *triple*
                           :format :concise)
```

```
<4: http://demo_news/12931 kb:containsPerson
     Barack Obama>
<12931 containsPerson Barack Obama>
TRIPLE-STORE-USER(24): (print-triple *triple*)
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
<12931 containsPerson Barack Obama>
```

Function **print-triple** prints a triple to standard output and returns the triple value in the short notation. We will see later in Section 2.2.6 how to create something like a database cursor for iterating through multiple triples that we find by querying a triple store. For now we will use query function **get-triples-list** that returns all triples matching a query in a list. The utility function **print-triples** prints all triples in a list:

```
TRIPLE-STORE-USER(27): (print-triples (list *triple*))
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
TRIPLE-STORE-USER(28): (print-triples (get-triples-list))
<http://demo_news/12931>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdfs#article> .
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
```

When **get-triples-list** is called with no arguments it simply returns all triples in a data store. We can specify query matching values for any combination of **:s**, **:p**, and **:o**. We can look at all triples that have their subject equal to the resource we created for the demo article:

```
TRIPLE-STORE-USER(31): (print-triples
                         (get-triples-list :s *demo-article*))
<http://demo_news/12931>
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
   <http://knowledgebooks.com/rdfs#article> .
<http://demo_news/12931>
   <http://knowledgebooks.com/rdfs#containsPerson>
   "Barack Obama" .
```

We can limit query results further; in this case we add the condition that the object must equal the value of the type **!kb:article**:

```
TRIPLE-STORE-USER(33): (print-triples
        (get-triples-list :s *demo-article*
                          :o !kb:article))
<http://demo_news/12931>
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
   <http://knowledgebooks.com/rdfs#article> .
```

I often need to manually reformat program example text and example program output in this book. The last three lines in the last example would appear on a single line if you are following along with these tutorial examples in a Lisp listener (as you should be!). In any case, RDF triple data in the NTriple format that we are using here is free-format: a triple is defined by three tokens (each with no embedded whitespace unless inside a string literal) and ended with a period character.

2.2.6. Using Cursors to Iterate Through Query Results

You are probably familiar with relational databases, the SQL query language, and client libraries that allow you to iterate through very large result sets. Allegrograph provides a cursor API for doing the same thing, as seen in this example:

```
TRIPLE-STORE-USER(39): (setq a-cursor (get-triples
                                        :s
                                        *demo-article*))
#<DB.AGRAPH::FILTERED-CURSOR
  #<DB.AGRAPH::ROW-CURSOR
    #<DB.AGRAPH::TRIPLE-RECORD-FILE @ #x113fd61a> ...
    #x11672082>
  @ #x1167219a>
TRIPLE-STORE-USER(40): (while (cursor-next-p a-cursor)
  ; cursor-next returns a vector, not a triple:
  (print (cursor-next-row a-cursor)))

<12931 type article>
<12931 containsPerson Barack Obama>
NIL
TRIPLE-STORE-USER(41):
```

I usually find it simpler to use the **get-triples-list** API that returns a list of results. I only use cursors when a query may return hundreds or thousands of results.

2.2.7. Saving Triple Stores to Disk as XML, N-Triples, and N3

It is often useful to copy either all triples in data store or triples matching a query to a flat disk file in N-Triples format:

```
(with-open-file (output "/tmp/sample.ntriple"
                        :direction :output
                        :if-does-not-exist :create)
     (print-triples (get-triples-list)
                    :stream output :format :ntriple))
```

In this example, I did not use any query filtering when calling **get-triples-list** so the entire contents of the data store is written to a local flat file. *Note that in this last example, everything gets read into memory; this could cause problems if you had millions of triples in the datastore.*

Output in the file might look like:

```
<http://demo_news/12931>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdfs#article> .
<http://demo_news/12931>
   <http://knowledgebooks.com/rdfs#containsPerson>
   "Barack Obama" .
```

Here we see two triples in N-Triple format. In most applications the RDF data store will be persistent and reused over multiple application restarts. While the disk-based triple store is persistent for many applications it is a good idea to support exporting triples in a standard format line the N-Triple format that we use here, the XML serialization format, or the newer and more compact N3 format.

2.3. AllegroGraph's Extensions to RDF

We have seen that RDF triples contain three values: $subject$, $predicate$, and $object$. We will cover this more in Chapter 3. AllegroGraph extends RDF adding two additional values:

1. graph-id – optional string to specify which graph the RDF triple belongs to
2. triple-id – unique triple ID

The *subject*, *predicate*, *object*, and *graph* value strings are uniquely stored in a global string table (like the symbol table a compiler uses) so that triples can more efficiently store indices rather than complete strings. Storing just a single copy of each unique string also save memory and disk storage. Comparing string table indices is also much faster than storing string values.

2.3.1. Examples Using Triple and Graph IDs

In the following example we will extend the example started earlier in this chapter by adding an additional triple specifying an optional graph ID value and the value for the RDF data store. If you had closed the connection to our example triple store with **(close-triple-store)** then start by reopening it:

```
(require :agraph)
(in-package :db.agraph.user)

(create-triple-store "/tmp/rdfstore_1")
;; default data store is kept in *db*
*db*
```

The value of ***db*** prints as:

```
#<DB.AGRAPH::TRIPLE-DB /tmp/rdfstore_1,
                       open @ #x11790d02>
```

After registering a namespace we add three triples. Unlike the examples seen earlier in this chapter, we specify values for two optional parameters for the connection and the graph value to function **add-triple**:

```
(register-namespace "kb"
                    "http://knowledgebooks.com/rdfs#")
(resource "http://demo_news/12931")
(defvar *demo-article*
        (resource "http://demo_news/12931"))

(add-triple *demo-article* !rdf:type !kb:article
            :db *db* :g !"news-data")
(add-triple *demo-article*
            !kb:containsPerson !"Barack Obama"
            :db *db* :g !"news-data")
(add-triple *demo-article* !kb:processed !"yes"
            :db *db* :g !"work-flow")
```

In addition to queries based on values of subject, predicate, and object we can also filter results by specifying a value for the graph:

```
;; query on optional graph value:
(print-triples (get-triples-list :g !"work-flow"))
```

producing the output:

```
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#processed>
  "yes" .
```

For the last three triples that we added to the triple store we used the optional **:db** keyword argument for function **add-triple**. Because we used the triple store stored in the global variable ***db*** using this optional keyword argument had no effect. However, it is possible to have multiple triple stores open at the same time so it can make sense to partition RDF data over multiple data stores on different servers. We will not concern ourselves in this book (except for mentioning it here) with AllegroGraph's client API functionality to access multiple distributed AllegroGraph servers. Franz's online documentation covers how to use the federation mechanism.

The function **add-triple** returns as its value the newly created triple's ID and has the side effect of adding the triple to the currently opened data store. While it is not best practice to use this unique internal AllegroGraph triple ID as a value referenced in another triple, there may be reasons in an application to store the IDs of newly created triples in order to be able to retrieve them from ID; for example:

```
TRIPLE-STORE-USER(15): (get-triple-by-id 3)
<12931 processed yes work-flow>
```

2.3.2. Support for Geo Location

Geo Location support in AllegroGraph is more general than 2D map coordinates or other 2D coordinate systems. I will briefly introduce you to the Geo Location APIs and also refer you to Franz's online documentation. The example code snippers for this section are found in the file quick_start_allegrograph_lisp_embedded/geoloc.lisp:

```
(require :agraph)

(in-package :db.agraph.user)
(enable-!-reader)
```

```
(register-namespace "g" "http://knowledgebooks.com/geo#")
(create-triple-store "/tmp/geospatial-test")

;; define some locations in Verde Valley, Arizona:
(defvar *locs*
  '(("Verde_Valley_Ranger_Station" 34.7666667 -112.1416667)
    ("Verde_Valley_School" 34.8047596 -111.8060388)
    ("Sedona" 34.8697222 -111.7602778)
    ("Cottonwood" 34.739 -112.009)
    ("Jerome" 34.75 -112.11)
    ("Flagstaff" 35.20 -111.63)
    ("Clarkdale" 34.76 -112.05)
    ("Mount_Wilson" 35.996 -114.611)
    ("Tuzigoot" 34.56 -111.84)))
```

Here I have defined a few locations in my area (in the mountains of Central Arizona) by latitude and longitude values. I will want to determine the minimum and maximum latitude and longitude in the data; the following simple map and reduce pattern does this:

```
(defvar *min-lat* (reduce #'min (mapcar #'cadr *locs*)))
(defvar *max-lat* (reduce #'max (mapcar #'cadr *locs*)))
(defvar *min-lon* (reduce #'min (mapcar #'caddr *locs*)))
(defvar *max-lon* (reduce #'max (mapcar #'caddr *locs*)))
```

The following code snippet creates and registers a new AllegroGraph Geo Spatial type based on the desired striping resolution and the minimum and maximum latitude and longitude values:

```
;; create a type:
(setf offset 5.0)
(flet ((fixup (num direction)
              (if (eq direction :min)
                  (- num offset)
                (+ num offset))))
  (setf *verde-valley-arizona*
        (db.agraph:register-latitude-striping-in-miles
         3
         :lat-min (fixup *min-lat* :min)
         :lat-max (fixup *max-lat* :max)
         :lon-min (fixup *min-lon* :min)
         :lon-max (fixup *max-lon* :max))))

(add-geospatial-subtype-to-db *verde-valley-arizona*)
```

After this setup we are ready to add latitude and longitude triples for each location:

```
(dolist (loc *locs*)
  (let ((name (intern-resource
         (format nil "http://knowledgebooks.com/geo#~a"
           (car loc)))))
    (print name)
    (add-triple name !g:isAt3
         (longitude-latitude->upi *verde-valley-arizona*
            (caddr loc) (cadr loc)))))

(index-all-triples :wait t)

(print
  (count-cursor
   (get-triples-haversine-miles *verde-valley-arizona*
                 !g:isAt3
                 -112.009 34.739 ; longitude and latitude
                30.0))       ; distance in miles

(dolist (distance '(50.0 30.0 10.0 5.0))
  (format t "~%~%Checking with distance = ~A~%" distance)
  (let ((cursor
         (get-triples-haversine-miles
            *verde-valley-arizona* !g:isAt3
            -112.009 34.739 distance)))
    (while (cursor-next-p cursor)
      (print (cursor-next-row cursor)))))
```

In this example, I print out the number locations in the triple store within 30 miles of the location (-112.009 34.739) and a list of all locations within 50, 30, 10, and 5 miles of this same location. Here is the part of the output for distances of 10 and 5 miles:

```
Checking with distance = 10.0

<Verde_Valley_Ranger_Station isAt3
                                +344559.99894-1120830.01273>
<Jerome isAt3 +344500-1120636.00212>
<Clarkdale isAt3 +344535.99394-1120300.01091>
<Cottonwood isAt3 +344420.39424-1120032.40939>

Checking with distance = 5.0
```

```
<Clarkdale isAt3 +344535.99394-1120300.01091>
<Cottonwood isAt3 +344420.39424-1120032.40939>
```

2.3.3. Support for Free Text Indexing

The AllegroGraph support for free text indexing is very useful and we will use later in this book in the example semantic web portal developed in Chapters 16 and 17. When I develop using Java or Ruby (the two languages I use most, in addition to Common Lisp) a common pattern is to use a data store like PostgreSQL or MongoDB with a separate text index and search library like Lucene. When working in Lisp with Allegrograph it is fast and agile to use Allegrograph for both data storage and text search. The example code snippets for this section are found in the file quick_start_allegrograph_lisp_embedded/text.lisp.

We will start with a new test triple store:

```
(require :agraph)
(in-package :db.agraph.user)

(enable-!-reader) ; enable the ! reader macro

(create-triple-store "/tmp/index_test")
(register-namespace "kb"
                    "http://knowledgebooks.com/rdfs#")
```

By default, text indexing is turned off on triples but we can request that all triples with a specified predicate will be indexed:

```
(register-freetext-predicate !kb:containsPerson)

(print (freetext-registered-predicates))
```

After registering a predicate, you can print out all predicates registered for indexing. The following code snippet creates a few test triples and all the triples using the predicate **!kb:containsPerson** will be indexed and thus searchable:

```
(resource "http://demo_news/12931")
(defvar *demo-article*
             (resource "http://demo_news/12931"))

(add-triple *demo-article* !rdf:type !kb:article)
(add-triple *demo-article* !kb:containsPerson
```

```
                              !"Barack Obama")
(add-triple *demo-article* !kb:containsPerson
                              !"Bill Clinton")
(add-triple *demo-article* !kb:containsPerson
                              !"Bill Jones")
```

The following uses the API **freetext-get-ids** that performs a free text search and returns all triple IDs that contain the query text; I then iterate over the results of a few additional example queries using cursors:

```
(print (freetext-get-ids "Clinton"))

(iterate-cursor (triple (freetext-get-triples
                         '(and "Bill" "Jones")))
    (print triple))
(iterate-cursor (triple (freetext-get-triples "Bill"))
    (print triple))
(iterate-cursor (triple (freetext-get-triples
                         '(or "Jones" "Clinton")))
    (print triple))
```

If I am not expecting many results for a text search query, then I prefer to use the API that returns all results at once in a list:

```
(print
 (freetext-get-triples-list '(or "Bill" "Barack")))
```

In this example I used the Lisp APIs for finding triples containing search terms. You will see in Chapter 5 how to use text search in SPARQL queries and in Chapter 9 I will show you how to use test search using Franz's Prolog query interface.

2.3.4. Comparing AllegroGraph With Other Semantic Web Frameworks

Although this book is about developing Semantic Web applications using AllegroGraph, it is also worthwhile to mention alternative technologies that can be used in addition to or instead of AllegroGraph.

The two alternative technologies that I have used most for Semantic Web applications are Swi-Prolog with its Semantic Web libraries (open source, LGPL) and the Java Sesame project (open source, BSD style license). Swi-Prolog is an excellent tool for experimenting and learning about the Semantic Web. Sesame is a complete Java

framework that is appropriate for applications written in Java. These alternatives have the advantage of being free to use but lack advantages of scalability and utility that a commercial product like AllegroGraph has.

2.4. AllegroGraph Quickstart Wrap Up

This short chapter gave you a brief introduction to running AllegroGraph interactively and some of the APIs that you will be using most frequently. This chapter has shown you the basics for using the Common Lisp APIs for AllegroGraph and if you have followed along with the examples here and then follow through the interactive SPARQL and Prolog examples in later chapters you will be able to understand and use the application specific examples from the last part of this book.

Part I.

Semantic Web Technologies

3. RDF

The Semantic Web is intended to provide a massive linked data set for use by software systems just as the World Wide Web provides a massive collection of linked web pages for human reading and browsing. The Semantic Web is like the World Wide Web in that anyone can generate any content that they want. This freedom to publish anything works for the web because we use our ability to understand natural language to interpret what we read – and often to dismiss material that based upon our own knowledge we consider to be incorrect.

The core concept for the Semantic Web is data integration and use from different sources. As we will soon see, the tools for implementing the Semantic Web are designed for encoding data and sharing data from many different sources.

The Resource Description Framework (RDF) is used to encode information and the RDF Schema (RDFS) language defines properties and classes and also facilitates using data with different RDF encodings without the need to convert data to use different schemas. For example, no need to change a property name in one data set to match the semantically identical property name used in another data set. Instead, you can add an RDF statement that states that the two properties have the same meaning.

I do not consider RDF data stores to be a replacement for relational databases but rather something that you will use with databases in your applications. RDF and relational databases solve difference problems. RDF is appropriate for sparse data representations that do not require inflexible schemas. You are free to define and use new properties and use these properties to make statements on existing resources. RDF offers more flexibility: defining properties used with classes is similar to defining the columns in a relational database table. You do not need to define properties for every instance of a class. This is analogous to a database table that can be missing columns for rows that do not have values for these columns (a sparse data representation). Furthermore, you can make ad hoc RDF statements about any resource without the need to update global schemas. We will use the SPARQL query language to access information in RDF data stores. SPARQL queries can contain optional matching clauses that work well with sparse data representations.

RDF data was originally encoded as XML and intended for automated processing. In this chapter we will use two simple to read formats called N-Triples and N3[1]. There

[1] N3 is a far better format to work with if you want to be able to read RDF data files and understand their contents. Currently AllegroGraph does not support N3 but Sesame does. I will usually use the

are many tools available that can be used to convert between all RDF formats so we might as well use formats that are easier to read and understand. RDF data consists of a set of triple values:

- subject - this is a URI
- predicate - this is a URI
- object - this is either a URI or a literal value

A statement in RDF is a triple composed of a subject, predicate, and object. A single resource containing a set of RDF triples can be referred to as an RDF graph. These resources might be a downloadable RDF file that you can load into AllegroGraph or Sesame, a web service that returns RDF data, or a SPARQL endpoint that is a web service that accepts SPARQL queries and returns information from an RDF data store.

While we tend to think in terms of objects and classes when using object oriented programming languages, we need to readjust our thinking when dealing with knowledge assets on the web. Instead of thinking about "objects" we deal with "resources" that are specified by URIs. In this way resources can be uniquely defined. We will soon see how we can associate different namespaces with URI prefixes – this will make it easier to deal with different resources with the same name that can be found in different sources of information.

While subjects will almost always be represented as URIs of resources, the object part of triples can be either URIs of resources or literal values. For literal values, the XML schema notation for specifying either a standard type like integer or string, or a custom type that is application domain specific.

You have probably read articles and other books on the Semantic Web, and if so, you are probably used to seeing RDF expressed in its XML serialization format: you will not see XML serialization in this book. Much of my own confusion when I was starting to use Semantic Web technologies ten years ago was directly caused by trying to think about RDF in XML form. RDF data is graph data and serializing RDF as XML is confusing and a waste of time when either the N-Triple format or even better, the N3 format are so much easier to read and understand.

Some of my work with Semantic Web technologies deals with processing news stories, extracting semantic information from the text, and storing it in RDF. I will use this application domain for the examples in this chapter. I deal with triples like:

- subject: a URI, for example the URL of a news article
- predicate: a relation like "a person's name" that is represented as a URI like

N3 format when discussing ideas but use the N-Triple format as input for example programs and for output when saving RDF data to files.

<http://knowledgebooks.com/rdf/person/name>[2]

- object: a literal value like ”Bill Clinton” or a URI

We will always use URIs[3] as values for subjects and predicates, and use URIs or string literals as values for objects. In any case URIs are usually preferred to string literals because they are unique; for example, consider the two possible values for a triple object:

- ”Bill Clinton” - as a string literal, the value may not refer to President Bill Clinton.
- <http://knowledgebooks.com/rdf/person#BillClinton> - as a URI, we can later make this URI a subject in a triple and use a relation to specify that this particular person had the job of President of the United States.

We will see an example of this preferred use but first we need to learn the N-Triple and N3 RDF formats.

3.1. RDF Examples in N-Triple and N3 Formats

In the Introduction I proposed the idea that RDF was more flexible than Object Modeling[4] in programming languages, relational databases, and XML with schemas[5]. If we can tag new attributes on the fly to existing data, how do we prevent what I might call “data chaos” as we modify existing data sources? It turns out that the solution to this problem is also the solution for encoding real semantics (or meaning) with data: we usually use unique URIs for RDF subjects, predicates, and objects, and usually with a preference for not using string literals. I will try to make this idea more clear with some examples.

Any part of a triple (subject, predicate, or object) is either a URI or a string literal. URIs encode namespaces. For example, the containsPerson property is used as the value of the predicate in this triple; the last example could properly be written as:

[2] URIs, like URLs, start with a protocol like HTTP that is followed by an internet domain.

[3] Uniform Resource Identifiers (URIs) are special in the sense that they (are supposed to) represent unique things or ideas. As we will see in Chapter 10, URIs can also be ”dereferenceable” in that we can treat them as URLs on the web and ”follow” them using HTTP to get additional information about a URI.

[4] We will model classes (or types) using RDFS and OWL but the difference is that an object in an OO language is explicitly declared to be a member of a class while a subject URI is considered to be in a class depending only on what properties it has. If we add a property and value to a subject URI then we may immediately change its RDFS or OWL class membership.

[5] I think that there is some similarity between modeling with RDF and document oriented data stores like MongoDB or CouchDB where any document in the system can have any attribute added at any time. This is very similar to being able to add additional RDF statements that either add information about a subject URI or add another property and value that somehow narrows the ”meaning” of a subject URI.

```
http://knowledgebooks.com/ontology/#containsPerson
```

The first part of this URI is considered to be the namespace[6] for (what we will use as a predicate) "containsPerson." Once we associate an abbreviation like **kb** for **http://knowledgebooks.com/ontology/** then we can just use the QName ("quick name") with the namespace abbreviation; for example:

```
kb:containsPerson
```

Being able to define abbreviation prefixes for namespaces makes RDF and RDFS files shorter and easier to read.

When different RDF triples use this same predicate, this is some assurance to us that all users of this predicate subscribe to the same meaning. Furthermore, we will see in Section 4.1 that we can use RDFS to state equivalency between this predicate (in the namespace http://knowledgebooks.com/ontology/) with predicates represented by different URIs used in other data sources. In an "artificial intelligence" sense, software that we write does not understand a predicate like "containsPerson" in the way that a human reader can by combining understood common meanings for the words "contains" and "person" but for many interesting and useful types of applications that is fine as long as the predicate is used consistently.

Because there are many sources of information about different resources the ability to define different namespaces and associate them with unique URI prefixes makes it easier to deal with situations.

A statement in N-Triple format consists of three URIs (or string literals – any combination) followed by a period to end the statement. While statements are often written one per line in a source file they can be broken across lines; it is the ending period which marks the end of a statement. The standard file extension for N-Triple format files is *.nt and the standard format for N3 format files is *.n3.

My preference is to use N-Triple format files as output from programs that I write to save data as RDF. I often use either command line tools or the Java Sesame library to convert N-Triple files to N3 if I will be reading them or even hand editing them. You will see why I prefer the N3 format when we look at an example:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China"  .
```

[6] You have seen me use the domain knowledgebooks.com several times in examples. I have owned this domain and used it for business since 1998 and I use it here for convenience. I could just as well use example.com. That said, the advantage of using my own domain is that I then have the flexibility to make this URI "dereferenceable" by adding an HTML document using this URI as a URL that describes what I mean by "containsPerson." Even better, I could have my web server look at the request header and return RDF data if the requested content type was "text/rdf"

Here we see the use of an abbreviation prefix "kb:" for the namespace for my company KnowledgeBooks.com ontologies. The first term in the RDF statement (the subject) is the URI of a news article. When we want to use a URL as a URI, we enclose it in angle brackets – as in this example. The second term (the predicate) is "containsCountry" in the "kb:" namespace. The last item in the statement (the object) is a string literal "China." I would describe this RDF statement in English as, "The news article at URI http://news.com/201234 mentions the country China."

This was a very simple N3 example which we will expand to show additional features of the N3 notation. As another example, suppose that this news article also mentions the USA. Instead of adding a whole new statement like this:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China"  .
<http://news.com/201234 /> kb:containsCountry "USA"  .
```

we can combine them using N3 notation. N3 allows us to collapse multiple RDF statements that share the same subject and optionally the same predicate:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China" ,
                                              "USA" .
```

We can also add in additional predicates that use the same subject:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234 /> kb:containsCountry "China" ,
                                              "USA" .
    kb:containsOrganization "United Nations" ;
    kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                      "Hu Jintao" , "George W. Bush" ,
                      "Pervez Musharraf" ,
                      "Vladimir Putin" ,
                      "Mahmoud Ahmadinejad" .
```

This single N3 statement represents ten individual RDF triples. Each section defining triples with the same subject and predicate have objects separated by commas and ending with a period. Please note that whatever RDF storage system we use (we will be using AllegroGraph) it makes no difference if we load RDF as XML, N-Triple, of N3 format files: internally subject, predicate, and object triples are stored in the same way and are used in the same way.

I promised you that the data in RDF data stores was easy to extend. As an example, let us assume that we have written software that is able to read online news articles and create RDF data that captures some of the semantics in the articles. If we extend our program to also recognize dates when the articles are published, we can simply reprocess articles and for each article add a triple to our RDF data store using the N-Triple format to set a publication date[7].

```
<http://news.com/2034 /> kb:datePublished "2008-05-11" .
```

Furthermore, if we do not have dates for all news articles that is often acceptable depending on the application.

3.2. The RDF Namespace

You just saw an example of using namespaces when I used my own namespace <http://knowledgebooks.com/ontology#>.

When you define a name space you can assign any "Quick name" (QName, or abbreviation) to the URI that uniquely identifies a namespace if you are using the N3 format.

The RDF namespace <http://www.w3.org/1999/02/22-rdf-syntax-ns#> is usually registered with the QName **rdf:** and I will use this convention. The next few sections show the definitions in the RDF namespace that I use in this book.

3.2.1. rdf:type

The **rdf:type** property is used to specify the type (or class) of a resource. Notice that we do not capitalize "type" because by convention we do not capitalize RDF property names. Here is an example in N3 format (with long lines split to fit the page width):

```
@prefix rdf:
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

@prefix
 kb:
 <http://knowledgebooks.com/rdf/publication#> .

<http://demo_news/12931> rdf:type kb:article .
```

[7] This example is pedantic since we can apply XML Scehma (XSL) data types to literal string values, this could be more accurately specified as "2008-05-11"@http://www.w3.org/2001/XMLSchema#date

Here we are converting the URL of a news web page to a resource and then defining a new triple that specifies the web page resource is or type kb:article (again, using the QName **kb:** for my knowledgebooks.com namespace).

3.2.2. rdf:Property

The **rdf:Property** class is, as you might guess from its name, used to describe and define properties. Notice that "Property" is capitalized because by convention we capitalize RDF class names.

This is a good place to show how we define new properties, using a previous example:

```
@prefix
 kbcontains:
 <http://knowledgebooks.com/rdf/contains#> .

<http://demo_news/12931>
  kbcontains:person
  "Barack Obama" .
```

I might make an additional statement about this URI stating that it is a property:

```
kbcontains:person rdf:type rdf:Property .
```

When we discuss RDF Schema (RDFS) in Chapter 4 we will see how to create sub-types and sub-properties.

3.3. Dereferenceable URIs

We have been using URIs as unique identifiers representing either physical objects (e.g., the moon), locations (e.g., London England), ideas or concepts (e.g., Christianity), etc. Additionally, a URI is dereferenceable if we can follow the URI with a web browser or software agent to fetch information from the URI. As an example, we often use the URI

```
http://xmlns.com/foaf/0.1/Person
```

to represent the concept of a person. This URI is dereferenceable because if we use a tool like wget or curl to fetch the content from this URI then we get an HTML document for the FOAF Vocabulary Specification. Dereferenceable content could also be a RDFS or OWL document describing the URI, a text document, etc.

3.4. RDF Wrap Up

If you read the World Wide Web Consortium's RDF Primer (highly recommended) at http://www.w3.org/TR/REC-rdf-syntax/ you will see many other classes and properties defined that, in my opinion, are often most useful when dealing with XML serialization of RDF. Using the N-Triple and N3 formats, I find that I usually just use rdf:type and rdf:Property in my own modeling efforts, along with a few identifiers defined in the RDFS namespace that we will look at in the next chapter.

An RDF triple has three parts: a subject, predicate, and object.[8] By itself, RDF is good for storing and accessing data but lacks functionality for modeling classes, defining properties, etc. We will extend RDF with RDF Schema (RDFS) in the next chapter.

[8] AllegroGraph also stores a unique integer triple ID and a graph ID for partitioning RDF data and to support graph operations. While using the triple ID and graph ID can be useful, my own preference is to stick with using just what is in the RDF standard.

4. RDFS

The World Wide Web Consortium RDF Schema (RDFS) definition can be read at http://www.w3.org/TR/rdf-schema/ and I recommend that you use this as a reference because I will discuss only the parts of RDFS that are required for implementing the examples in this book. The RDFS namespace http://www.w3.org/2000/01/rdf-schema# is usually registered with the QName **rdfs:** and I will use this convention[1].

4.1. Extending RDF with RDF Schema

RDFS supports the definition of classes and properties based on set inclusion. In RDFS classes and properties are orthogonal. We will not simply be using properties to define data attributes for classes – this is different than object modeling and object oriented programming languages like Java. RDFS is encoded as RDF – the same syntax.

Because the Semantic Web is intended to be processed automatically by software systems it is encoded as RDF. There is a problem that must be solved in implementing and using the Semantic Web: everyone who publishes Semantic Web data is free to create their own RDF schemas for storing data; for example, there is usually no single standard RDF schema definition for topics like news stories, stock market data, people's names, organizations, etc. Understanding the difficulty of integrating different data sources in different formats helps to understand the design decisions behind the Semantic Web: the designers wanted to make it not only possible but also easy to use data from different sources that might use similar schema to define properties and classes. One common usage pattern is using RDFS to define two properties that both define a person's last name have the same meaning and that we can combine data that use different schema.

We will start with an example that also uses dRDFS an is an extension of the example in the last section. After defining **kb:** and **rdfs:** namespace QNames, we add a few additional RDF statements (that are RDFS):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
```

[1] The actual namespace abbreviations that you use have no effect as long as you consistently use whatever QName values you set for URIs in the RDF statements that use the abbreviations.

```
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .
```

The last three lines that are themselves valid RDF triples declare that:

- The property containsCity is a subproperty of containsPlace.
- The property containsCountry is a subproperty of containsPlace.
- The property containsState is a subproperty of containsPlace.

Why is this useful? For at least two reasons:

- You can query an RDF data store for all triples that use property containsPlace and also match triples with property equal to containsCity, containsCountry, or containsState. There may not even be any triples that explicitly use the property containsPlace.
- Consider a hypothetical case where you are using two different RDF data stores that use different properties for naming cities: "cityName" and "city." You can define "cityName" to be a subproperty of "city" and then write all queries against the single property name "city." This removes the necessity to convert data from different sources to use the same Schema.

In addition to providing a vocabulary for describing properties and class membership by properties, RDFS is also used for logical inference to infer new triples, combine data from different RDF data sources, and to allow effective querying of RDF data stores. We will see examples of more RDFS features in Chapter 5 when we perform SPARQL queries.

4.2. Modeling with RDFS

While RDFS is not as expressive of a modeling language as the RDFS++[2] or OWL, the combination of RDF and RDFS is usually adequate for many semantic web applications. Reasoning with and using more expressive modeling languages will require increasingly more processing time. Combined with the simplicity of RDF and RDFS it is a good idea to start with less expressive and only "move up the expressivity scale" as needed.

[2]RDFS++ is a Franz extension to RDFS that adds some parts of OWL. I cover RDFS++ in some detail in the Lisp Edition of this book and mention some aspects of RDFS++ in Section 4.3, the Java Edition.

Here is a short example on using RDFS to extend RDF (assume that my namespace **kb:** and the RDFS namespace **rdfs:** are defined):

```
kb:Person rdf:type rdfs:Class .
kb:Person rdfs:comment "represents a human" .
kb:Manager rdf:type kb:Person .
kb:Manager rdfs:domain kb:Person .
kb:Engineer rdf:type kb:Person .
kb:Engineer rdfs:domain kb:Person .
```

Here we see the use of **rdfs:comment** used to add a human readable comment to the new class **kb:Person**. When we define the new classes **kb:Manager** and **kb:Engineer** we make them subclasses of **kb:Person** instead of the top level super class **rdfs:Class**. We will look at examples later in that that demonstrate the utility of models using class hierarchies and hierarchies of properties – for now it is enough to introduce the notation.

The **rdfs:domain** of an **rdf:property** specifies the class of the subject in a triple while **rdfs:range** of an **rdf:property** specifies the class of the object in a triple. Just as strongly typed programming languages like Java help catch errors by performing type analysis, creating (or using existing) good RDFS property and class definitions helps RDFS, RDFS++, and OWL descriptive logic reasoners to catch modeling and data definition errors. These definitions also help reasoning systems infer new triples that are not explicitly defined in a triple data store.

We continue the current example by adding property definitions and then asserting a triple that is valid given the type and property restrictions that we have defined using RDFS:

```
kb:supervisorOf rdfs:domain kb:Manager .
kb:supervisorOf rdfs:range kb:Engineer .

"Mary Johnson" rdf:type kb:Manager .
"John Smith'' rdf:type kb:Engineer .

"Mary Johnson" kb:supervisorOf "John Smith" .
```

If I tried to add a triple with “Mary Johnson” and “John Smith” reversed in the last RFD statement then an RDFS inference/reasoning system could catch the error. This example is not ideal because I am using string literals as the subjects in triples. In general, you probably want to define a specific namespace for concrete resources representing entities like the people in this example.

The property **rdfs:subClassOf** is used to state that all instances of one class are also instances of another class. The property **rdfs:subPropertyOf** is used to state that

all resources related by one property are also related by another; for example, given the following N3 statements that use string literals as resources to make this example shorter:

```
kb:familyMember rdf:type rdf:Property .
kb:ancestorOf rdf:type rdf:Property .
kb:parentOf rdf:type rdf:Property .

kb:ancestorOf rdfs:subPropertyOf kb:familyMember .
kb:parentOf rdfs:subPropertyOf kb:ancestorOf .

"Marry Smith" kb:parentOf "Sam" .
```

then the following is valid:

```
"Marry Smith" kb:ancestorOf "Sam" .
"Marry Smith" kb:familyMember "Sam" .
```

We have just seen that a common use of RDFS is to define additional application or data-source specific properties and classes in order to express relationships between resources and the types of resources. Whenever possible you will want to reuse existing RDFS properties and resources that you find on the web. For instance, in the last example I defined my own subclass **kb:person** instead of using the Friend of a Friend (FOAF) namespace's definition of person. I did this for pedantic reasons: I wanted to show you how to define your own classes and properties.

4.3. AllegroGraph RDFS++ Extensions

The *unofficial* version of RDFS/OWL called RDFS++ is a practical compromise between DL OWL and RDFS inferencing. AllegroGraph supports the following predicates:

- rdf:type – discussed in Chapter 3
- rdf:property – discussed in Chapter 3
- rdfs:subClassOf – discussed in Chapter 4
- rdfs:range – discussed in Chapter 4
- rdfs:domain – discussed in Chapter 4
- rdfs:subPropertyOf – discussed in Chapter 4

- owl:sameAs
- owl:inverseOf
- owl:TransitiveProperty

We will now discuss **owl:sameAs**, **owl:inverseOf**, and **owl:TransitiveProperty** to complete the discussion of frequently used RDFS predicates seen earlier in this Chapter.

4.3.1. owl:sameAs

If the same entity is represented by two distinct URIs **owl:sameAs** can be used to assert that the URIs refer to the same entity. For example, two different knowledge sources might might define different URIs in their own namespaces for President Barack Obama. Rather than translate data from one knowledge source to another it is simpler to equate the two unique URIs. For example:

```
kb:BillClinton rdf:type kb:Person .
kb:BillClinton owl:sameAs mynews:WilliamClinton
```

Then the following can be verified using an RDFS++ or OWL DL capable reasoner:

```
mynews:WilliamClinton rdf:type kb:Person .
```

4.3.2. owl:inverseOf

We can use **owl:inverseOf** to declare that one property is the inverse of another.

```
:parentOf owl:inverseOf :childOf .
"John Smith" :parentOf "Nellie Smith" .
```

There is something new in this example: I am using a "default namespace" for **:parentOf** and **:childOf**. A default namespace is assumed to be application specific and that no external software agents will need access to resources defined in the default namespace.

Given the two previous RDF statements we can infer that the following is also true:

```
"Nellie Smith" :childOf "John Smith" .
```

4.3.3. owl:TransitiveProperty

As its name implies **owl:TransitiveProperty** is used to declare that a property is transitive as the following example shows:

```
kb:ancestorOf a rdf:Property .
"John Smith"  kb:ancestorOf "Nellie Smith" .
"Nellie Smith" kb:ancestorOf "Billie Smith" .
```

There is something new in this example: in N3 you can use **a** as shorthand for **rdf:type**. Given the last three RDF statements we can infer that:

```
"John Smith" : kb:ancestorOf "Billie Smith" .
```

4.4. RDFS Wrapup

I find that RDFS provides a good compromise: it is simpler to use than the Web Ontology Language (OWL) and is expressive enough for many linked data applications. As we have seen, AllegroGraph supports RDFS++ which is RDFS with a few OWL extensions:

1. rdf:type
2. rdfs:subClassOf
3. rdfs:domain
4. rdfs:range
5. rdfs:subPropertyOf
6. owl:sameAs
7. owl:inverseOf
8. owl:TransitiveProperty

Since I only briefly covered these extensions you may want to read the documentation on Franz's web site[3].

Sesame supports RDFS "out of the box" and back end reasoners are available for Sesame that support OWL[4]. Sesame is likely to have OWL reasoning built in to the

[3] http://www.franz.com/agraph/support/learning/Overview-of-RDFS++.html

[4] You can download SwiftOWLIM or BigOWLIM at http://www.ontotext.com/owlim/ and use either as a SAIL backend repository to get OWL reasoning capability.

standard distribution in the future. My advice is to start building applications with RDF and RDFS with a view to using OWL as the need arises. If you are using AllegroGraph for your application development then certainly use the RDFS++ extensions if RDFS is too limited for your applications.

We have been using SPARQL in examples and in the next chapter we will look at SPARQL in some detail.

5. The SPARQL Query Language

SPARQL is a query language used to query RDF data stores. While SPARQL may initially look like SQL you will see that there are important differences because the data is graph-based so queries match graph patterns instead SQL's relational matching operations. So the syntax is similar but SPARQL queries graph data and SQL queries relational data in tables.

We have already been using SPARQL queries in examples in this book. I will give you more introductory material in this chapter before using SPARQL in larger example programs later in this book.

5.1. Example RDF Data in N3 Format

We will use the N3 format RDF file data/news.n3 for examples in this chapter. We use the N3 format because it is easier to read and understand. There is an equivalent N-Triple format file data/news.nt because AllegroGraph does not currently support loading N3 files. I created these files automatically by spidering Reuters news stories on the news.yahoo.com web site and automatically extracting named entities from the text of the articles. I used the Java Sesame library to convert the generated N-Triple file to N3 format. We will see similar techniques for extracting named entities from text in Chapter 11 when I develop utilities for using the Reuters Open Calais web services. We will also use my Natural Language Processing (NLP) library in Chapter 12 to do the same thing. In this chapter we use these sample RDF files that I have created using Open Calais and news articles that I found on the web.

You have already seen snippets of this file in Section 4.1 and I list the entire file here for reference ,edited to fit line width. You may find the file news.n3 easier to read if you are at your computer and open the file in a text editor so you will not be limited to what fits on a book page):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
```

```
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .

kb:containsState rdfs:subPropertyOf kb:containsPlace .

<http://yahoo.com/20080616/usa_flooding_dc_16 />
    kb:containsCity "Burlington" , "Denver" ,
                    "St. Paul" ," Chicago" ,
                    "Quincy" , "CHICAGO" ,
                    "Iowa City" ;
    kb:containsRegion "U.S. Midwest" , "Midwest" ;
    kb:containsCountry "United States" , "Japan" ;
    kb:containsState "Minnesota" , "Illinois" ,
                     "Mississippi" , "Iowa" ;
    kb:containsOrganization "National Guard" ,
                     "U.S. Department of Agriculture" ,
                     "White House" ,
                     "Chicago Board of Trade" ,
                     "Department of Transportation" ;
    kb:containsPerson "Dena Gray-Fisher" ,
                      "Donald Miller" ,
                      "Glenn Hollander" ,
                      "Rich Feltes" ,
                      "George W. Bush" ;
    kb:containsIndustryTerm "food inflation" , "food" ,
                            "finance ministers" ,
                            "oil" .

<http://yahoo.com/78325/ts_nm/usa_politics_dc_2 />
    kb:containsCity "Washington" , "Baghdad" ,
                    "Arlington" , "Flint" ;
    kb:containsCountry "United States" ,
                       "Afghanistan" ,
                       "Iraq" ;
    kb:containsState "Illinois" , "Virginia" ,
                     "Arizona" , "Michigan" ;
    kb:containsOrganization "White House" ,
                            "Obama administration" ,
                            "Iraqi government" ;
    kb:containsPerson "David Petraeus" ,
                      "John McCain" ,
                      "Hoshiyar Zebari" ,
                      "Barack Obama" ,
                      "George W. Bush" ,
                      "Carly Fiorina" ;
    kb:containsIndustryTerm "oil prices" .
```

```
<http://yahoo.com/10944/ts_nm/worldleaders_dc_1 />
    kb:containsCity "WASHINGTON" ;
    kb:containsCountry "United States" , "Pakistan" ,
                       "Islamic Republic of Iran" ;
    kb:containsState "Maryland" ;
    kb:containsOrganization "University of Maryland" ,
                            "United Nations" ;
    kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                      "Hu Jintao" , "George W. Bush" ,
                      "Pervez Musharraf" ,
                      "Vladimir Putin" ,
                      "Steven Kull" ,
                      "Mahmoud Ahmadinejad" .

<http://yahoo.com/10622/global_economy_dc_4 />
    kb:containsCity "Sao Paulo" , "Kuala Lumpur" ;
    kb:containsRegion "Midwest" ;
    kb:containsCountry "United States" , "Britain" ,
                       "Saudi Arabia" , "Spain" ,
                       "Italy" , India" ,
                       ""France" , "Canada" ,
                       "Russia" , "Germany" , "China" ,
                       "Japan" , "South Korea" ;
    kb:containsOrganization "Federal Reserve Bank" ,
                            "European Union" ,
                            "European Central Bank" ,
                            "European Commission" ;
    kb:containsPerson "Lee Myung-bak" , "Rajat Nag" ,
                      "Luiz Inacio Lula da Silva" ,
                      "Jeffrey Lacker" ;
    kb:containsCompany "Development Bank Managing" ,
                       "Reuters" ,
                       "Richmond Federal Reserve Bank" ;
    kb:containsIndustryTerm "central bank" , "food" ,
                            "energy costs" ,
                            "finance ministers" ,
                            "crude oil prices" ,
                            "oil prices" ,
                            "oil shock" ,
                            "food prices" ,
                            "Finance ministers" ,
                            "Oil prices" , "oil" .
```

5.2. Example SPARQL SELECT Queries

In the following examples, we will look at queries but not the results. You have already seen results of SPARQL queries when we ran the AllegroGraph and Sesame wrapper examples.

We will start with a simple SPARQL query for subjects (news article URLs) and objects (matching countries) with the value for the predicate equal to $containsCountry$:

```
SELECT ?subject ?object
  WHERE {
    ?subject
    http://knowledgebooks.com/ontology#containsCountry>
    ?object .
  }
```

Variables in queries start with a question mark character and can have any names. Since we are using two free variables (*?subject* and *?object*) each matching result will contain two values, one for each of these variables.

We can make this last query easier to read and reduce the chance of misspelling errors by using a namespace prefix:

```
PREFIX kb:  <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
  WHERE {
      ?subject kb:containsCountry ?object .
  }
```

We could have filtered on any other predicate, for instance *containsPlace*. Here is another example using a match against a string literal to find all articles exactly matching the text "Maryland."

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject
WHERE { ?subject kb:containsState "Maryland" . }
```

We can also match partial string literals against regular expressions:

```
PREFIX kb:
SELECT ?subject ?object
```

```
   WHERE {
     ?subject
     kb:containsOrganization
     ?object FILTER regex(?object, "University") .
   }
```

Prior to this last example query we only requested that the query return values for subject and predicate for triples that matched the query. However, we might want to return all triples whose subject (in this case a news article URI) is in one of the matched triples. Note that there are two matching triples, each terminated with a period:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject ?a_predicate ?an_object
   WHERE {
    ?subject
    kb:containsOrganization
    ?object FILTER regex(?object, "University") .

    ?subject ?a_predicate ?an_object .
   }
DISTINCT
ORDER BY ?a_predicate  ?an_object
LIMIT 10
OFFSET 5
```

When WHERE clauses contain more than one triple pattern to match, this is equivalent to a Boolean "and" operation. The DISTINCT clause removes duplicate results. The ORDER BY clause sorts the output in alphabetical order: in this case first by predicate (containsCity, containsCountry, etc.) and then by object. The LIMIT modifier limits the number of results returned and the OFFSET modifier sets the number of matching results to skip.

We are finished with our quick tutorial on using the SELECT query form. There are three other query forms that I will now briefly[1] cover:

- CONSTRUCT – returns a new RDF graph of query results
- ASK – returns Boolean true or false indicating if a query matches any triples
- DESCRIBE – returns a new RDF graph containing matched resources

[1] I almost always use just SELECT queries in applications.

5.3. Example SPARQL CONSTRUCT Queries

A SPARQL CONSTRUCT query acts like a SELECT query in that part of an RDF graph is matched. For CONSTRUCT queries, the matching subgraph is returned.

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
CONSTRUCT {kb:StateOfMaryland kb:isDiscussedIn ?subject }
WHERE { ?subject kb:containsState "Maryland" . }
```

The output graph would only contain one RDF statement because only one of our test news stories mentioned the state of Maryland:

```
kb:StateOfMaryland
  kb:isDiscussedIn
  <http://yahoo.com/10944/ts_nm/worldleaders_dc_1 /> .
```

5.4. Example SPARQL ASK Queries

SPARQL ask queries check the validity of an RDF statement (possibly including variables) and returns ”yes” or ”no” as the query result. In a similar example to the CONSTRUCT query, here I ask if there are any articles that discuss the state of Maryland:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
ASK { ?subject kb:containsState "Maryland"  }
```

5.5. Example SPARQL DESCRIBE Queries

Currently the SPARQL standard leaves the output from DESCRIBE queries as only partly defined and implementaton specific. A DESCRIBE query is similar to a CONSTRUCT query in that it returns information about resources in queries. The following example should return a graph showing information of all triples using the resource matched by the variable ?subject:

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
DECRIBE ?subject
WHERE { ?subject kb:containsState "Maryland" . }
```

5.6. Wrapup

This chapter ends the background material on Semantic Web Technologies. The remaining chapters in this book will be examples of gathering useful linked data and using it in applications.

6. RDFS++ and OWL

There are three standard versions of OWL: Lite, Description Logic (DL), and Full.

Because more expressive versions of OWL require more computing resources (and OWL Full may often require too many resources to be useful except for small problems) it is usually a good idea to use the minimum modeling constructs when develping a Semantic Web application.

OWL DL strikes a good balance between expressiveness and computability. OWL DL reasoners are usually complete (that is, they provide all possible answers to queries). The problem in many real-world applications is the unpredictability of how long a query will take to execute.

While the three versions of OWL are standards there is an ad-hoc definition called RDFS++ that is more expressive that RDF + RDFS but less expressive than OWL. Since AllegroGraph supports RDFS++ but not OWL (without using external reasoning systems) we will not cover OWL constructs in this book unless they are implemented in RDFS++.

6.1. Properties Supported In RDFS++

The *unofficial* version of RDFS/OWL called RDFS++ is a practical compromise between DL OWL and RDFS inferencing. AllegroGraph supports the following predicates, the first six have already been discussed in Chapters 3 and 4:

- rdf:type – discussed in Chapter 3
- rdf:property – discussed in Chapter 3
- rdfs:subClassOf – discussed in Chapter 4
- rdfs:range – discussed in Chapter 4
- rdfs:domain – discussed in Chapter 4
- rdfs:subPropertyOf – discussed in Chapter 4
- owl:sameAs

- owl:inverseOf
- owl:TransitiveProperty

We will now discuss **owl:sameAs**, **owl:inverseOf**, and **owl:TransitiveProperty**. We will see in Chapters 7 and 8 interactive examples of these predicates.

6.1.1. owl:sameAs

If the same entity is represented by two distinct URIs **owl:sameAs** can be used to assert that the URIs refer to the same entity. For example, two different knowledge sources might might define different URIs in their own namespaces for President Bill Clinton. Rather than translate data from one knowledge source to another it is simpler to equate the two unique URIs. For example:

```
kb:BillClinton rdf:type kb:Person .
kb:BillClinton owl:sameAs mynews:WilliamClinton
```

Then the following can be verified using a RDFS++ or OWL DL capable reasoner:

```
mynews:WilliamClinton rdf:type kb:Person .
```

6.1.2. owl:inverseOf

We can use **owl:inverseOf** to declare that one property is the inverse of another.

```
:parentOf owl:inverseOf :childOf .
"John Smith" :parentOf "Nellie Smith" .
```

There is something new in this example: I am using a "default namespace" for **:parentOf** and **:childOf**. A default namespace is assumed to be application specific and that no external software agents will need access to resources defined in the default namespace.

Given the two previous RDF statements we can infer that the following is also true:

```
"Nellie Smith" :childOf "John Smith" .
```

6.1.3. owl:TransitiveProperty

As its name implies **owl:TransitiveProperty** is used to declare that a property is transitive as the following example shows:

```
kb:ancestorOf a owl:TransitiveProperty .
"John Smith"  kb:ancestorOf "Nellie Smith" .
"Nellie Smith" kb:ancestorOf "Billie Smith" .
```

There is something new in this example: in N3 you can use **a** as shorthand for **rdf:type**. Given the last three RDF statements we can infer that:

```
"John Smith" : kb:ancestorOf "Billie Smith" .
```

6.2. RDF, RDFS, and RDFS++ Modeling Wrap Up

You should now be getting the idea that RDF, RDFS, and RDFS++ modeling is quite different than what you have experienced in object modeling in object oriented software development and also different than modeling data using relational algebra.

The first difference lies in the ability to extend models by adding new properties and new statements, usually without invalidating previous statements. For example, if you have RDF data encoding information about a group of people using properties like email address, name, and telephone number then you can also add statements later using properties like ”knows person” or ”street address” without invalidating previously defined information.

The second difference is the ability to infer information that is not explicitly encoded as RDF triples.

Part II.

AllegroGraph Extended Tutorial

7. SPARQL Queries Using AllegroGraph APIs

We saw some example SPARQL queries in Chapter 5 where we expressed the queries in text form. In this chapter we will work through SPARQL examples using snippets of Lisp code and the AllegroGraph APIs. We will see more interactive examples that are built on the examples in this chapter when we look at more reasoning examples in Chapter 8 and AllegroGraph's Prolog interface in Chapter 9.

The file sparql/news.nt was generated automatically by spidering a list of Reuters news articles on Yahoo News and using the Open Calais entity extraction web services that we will discuss in some detail in Chapter 11. This generated N-Triple file does not use name space abreiviations as you can see from the first line in the file:

```
<http://news.yahoo.com/20080616/ts_nm/usa_flooding_dc_16 />
 <http:://knowledgebooks.com/ontology#containsCity>
 "Burlington" .
```

7.1. Using Namespaces

We have seen in earlier chapters how we use RDF triples from different sources that we identify as belonging to different namespaces. The function **register-namespace** is used to associate quick name abbreviations with namespace URIs.

AllegroGraph does not support reading the concise N3 format that we used in the last chapter but we can make the N-Triple data file easier to work with by copying it to the file news_ns.nt and using edit macros to convert to using the namespaces:

```
(register-namespace
  "kb"
  "http:://knowledgebooks.com/ontology#")
(register-namespace
  "test_news"
  "http://news.yahoo.com/s/nm/20080616/ts_nm")
```

7.2. Reading RDF Data From Files

We created new RDF triples programatically in Chapter 2. In this section we will see how to read triple stores from disk.

In the last section we registered the namespace **test_news** that is used in the first line of the edited file news_ns.nt:

```
!test_news:usa_flooding_dc_16
  !kb:containsCity
  "Burlington" .
```

The file sparql/sparql_query.lisp contains all of the examples in this chapter. I encourage you to read through this chapter as well as the next two AllegroGraph tutorial chapters with a Lisp listener window open and try all examples for yourself and then experiment with the techniques we cover in the text.

We need to load this N-Triple file before performing any queries:

```
(load-ntriples "news.nt")
```

This load operation will fail if you have not defined the namespaces used in the file with the function **register-namespace**.

7.3. Lisp APIs for Queires

If we try a query the default is to return RDF in XML format (and we agreed to not use XML encodings!) If you are following this tutorial interactively, try evaluating the following expression:

```
(sparql:run-sparql "
  PREFIX kb: <http://knowledgebooks.com/ontology#>
  SELECT ?article_uri ?city_name  WHERE {
    ?article_uri kb:containsCity ?city_name .
  }")
```

Fortunately we can use optional arguments on the **sparql:run-sparql** function to get a more convenient “Lisp like” return values for SPARQL queries. This example gets the results as a list of hash tables:

```
(defvar *r1*
 (sparql:run-sparql "
   PREFIX kb: <http:://knowledgebooks.com/ontology#>
   SELECT ?article_uri ?city_name  WHERE {
          ?article_uri kb:containsCity ?city_name .
   }"
  :results-format :hashes))

(dolist (result *r1*)
  (maphash
   #'(lambda (key value)
       (format t " key: ~S~% value: ~S~%~%"
               key value))
   result))
```

The output from this code snippet is:

```
 key: |?article_uri|
 value: {http://news.yahoo.com/...}

 key: |?city_name|
 value: {Burlington}
;; etc.
```

The SPARQL **ASK** command checks to see if a given query produces any results. The following example request a Lisp true/false return value to the question "Does any article contain the city Chicago?":

```
(sparql:run-sparql "
  PREFIX kb: <http:://knowledgebooks.com/ontology#>
  ASK  {
    ?any_article kb:containsCity 'Chicago'
  }"
  :results-format :boolean)
```

There are many possible options for the **:results-format** keyword argument, including:

- :sparql-xml – serializes the results as XML to output-stream
- :sparql-json – serializes the results as JSON data to output-stream
- :sparql-ttl – serializes the results as Turtle encoding to output-stream (Turtle is a simplified version of N3, like N-Triples with namespaces)

- :hashes – returns a list of hash tables (as seen in a previous example)
- :arrays – returns a list of arrays for each results
- :lists – returns a list of lists for each results
- :count – returns an integer for the number of results

At some loss of efficiency it is sometimes useful to match string values against regular expressions; for example:

```
(sparql:run-sparql "
  PREFIX kb: <http:://knowledgebooks.com/ontology#>
  SELECT ?article_uri  WHERE {
    ?article_uri kb:containsPerson ?person_name .
    FILTER regex(?person_name, '^*Putin*')
  }"
  :results-format :lists)

;; output:
(({http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders /}))
```

7.4. Wrap Up

You learned how to query RDF triples in a repository using the Lisp AllegroGraph APIs in this chapter. We only considered triples that we explicitly added to the triple store and in later chapters we will automate the collection of data from the Internet and convert it to RDF and add it to a local triple store for reuse. Much of the power of Semantic Web technologies in general and AllegroGraph in particular is the ability to use triples that are inferred from RDFS without being explicitly created. This capability is covered in the next chapter in addition to techniques for using different data sources implemented using different schemas.

8. AllegroGraph Reasoning System

In the last chapter we saw how SPARQL queries can be used to find specific data in an RDF graph. So far we have only seen examples of finding data that has been explicitly added to an RDF data repository.

However RDFS, RDFS++, and OWL reasoners can return results that are known implicitly by using inference. We have already seen that AllegroGraph supports reasoning using the following predicates that can be used to infer new relationships that are not explicitly stated in the RDF data stored in AllegroGraph:

- rdf:type – discussed in Chapter 3
- rdf:property – discussed in Chapter 3
- rdfs:subClassOf – discussed in Chapter 4
- rdfs:range – discussed in Chapter 4
- rdfs:domain – discussed in Chapter 4
- rdfs:subPropertyOf – discussed in Chapter 4
- owl:sameAs – discussed in Chapter 6
- owl:inverseOf– discussed in Chapter 6
- owl:TransitiveProperty – discussed in Chapter 6

8.1. Enabling RDFS++ Reasoning on a Triple Store

We will look at the AllegroGraphs APIs and programming techniques for reasoning in detail in this chapter. By default AllegroGraph triple stores do not support RDFS++ reasoning. You must enable RDFS++ reasoning functionality by:

```
(apply-rdfs++-reasoner :db *db*)
```

This function works via side effect: the specified data store is converted to support inferencing. Since the default database ***db*** can be assumed, this can be shortened to:

```
(apply-rdfs++-reasoner)
```

If you use multiple data stores at the same time you can use different inference support for each. The remainder of this chapter uses reasoning to infer[1] new information.

8.2. Inferring New Triples: rdf:type vs. rdfs:subClassOf Example

In the following example, we define two triples and then perform a SPARQL query that answers a question based on a new inferred triple that has not been explicitly added to the triple store:

```
(add-triple !kb:man !rdfs:type !kb:person)
(add-triple !kb:sam !rdf:type !kb:man)

(sparql:run-sparql "
  PREFIX kb: <http:://knowledgebooks.com/ontology#>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  ASK  {
    kb:sam rdf:type kb:man
  }"
  :results-format :boolean)
```

This query returns a Lisp true value **T**. You might think that since **kb:man** is declared of **rdf:type kb:person** that the following query would return a true value:

```
(add-triple !kb:man !rdf:type !kb:person)
(add-triple !kb:sam !rdf:type !kb:man)
```

[1] Implementations of RDF triple stores that support RDFS, RDFS++, or OWL reasoning can implement inferred triples in different ways. One approach is to "pre-calculate" inferred triples using forward chaining inference; this approach is used by the Sesame library. A different approach used in Allegrograph is to infer triples at query time. The results should (hopefully) be the same.

```
(sparql:run-sparql "
  PREFIX kb: <http://knowledgebooks.com/ontology#>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  ASK  {
    kb:sam rdf:type kb:person
  }"
  :results-format :boolean)
```

This, however, returns a Lisp false value **NIL**. To get what you probably thought was the expected subclass behavior we can use **rdfs:subClassOf**:

```
(add-triple !kb:man !rdfs:subClassOf !kb:person)
```

Now the last query returns a true (Lisp **T**) value.

8.3. Using Inverse Properties

Properties define a one-way relationship between resources. Sometimes a property like ”husband of” has an inverse property like ”wife of” so when we say that Mark is the husband of Carol we would like an automatic logical inference that Carol is the wife of Mark is true also.

```
(require :agraph)
(in-package :db.agraph.user)agraph.user)
(create-triple-store "/tmp/rdfstore_2")
(register-namespace "kb" "http://knowledgebooks.com/ontology#")
(apply-rdfs++-reasoner)
(enable-!-reader)
(add-triple !kb:Mark !kb:husband-of !kb:Carol)
(add-triple !kb:wife-of !owl:inverseOf !kb:husband-of)
```

We can now infer who **wife-of** relationships:

```
(sparql:run-sparql "
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?y ?x WHERE { ?y kb:wife-of ?x }")
```

Since I did not specify the output data format, the default is RDF encoded as XML:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
  <head>
    <variable name="y"/>
    <variable name="x"/>
  </head>
  <results>
    <result>
      <binding name="y">
        <uri>http::://knowledgebooks.com/ontology#Carol</uri>
      </binding>
      <binding name="x">
        <uri>http::://knowledgebooks.com/ontology#Mark</uri>
      </binding>
    </result>
  </results>
</sparql>
```

I find other output formats generally easier to use; for example specifying **:results-format :lists** yields:

```
(({Carol} {Mark}))
```

and specifying **:results-format :hashes** like:

```
(sparql:run-sparql "
PREFIX kb: <http::://knowledgebooks.com/ontology#>
SELECT ?y ?x WHERE { ?y kb:wife-of ?x }" :results-format :hashes
```

will yield:

Specifying an output format of **:results-format :sparql-json** yields:

```
{
  "head" : {
    "vars" : ["y", "x"]
  },
  "results" : {
    "bindings" : [
```

```
      {
         "y":{"type":"uri",
              "value":
                "http:://knowledgebooks.com/ontology#Carol"},
         "x":{"type":"uri",
              "value":
                "http:://knowledgebooks.com/ontology#Mark"}
      }
    ]
  }
}
```

Yet another output option is **:results-format :alists**

8.4. Using the Same As Property

It is often useful to make an RDF statement that two different resources are equivalent as in this example:

```
(add-triple !kb:Mark !kb:name !kb:"Mark Watson")
(register-namespace "test_news" "http://news.yahoo.com/s/nm/20080
(add-triple !test_news:Mark !kb:height !"6 feet 4 inches")
(add-triple !kb:mark !owl:sameAs !kb:test_news:Mark)
(apply-rdfs++-reasoner)
```

You will be surprised to see only the following results:

```
(({name} {Mark Watson})
 ({sameAs} {test_news:Mark}))
```

With just RDFS++ reasoning, the height of Mark will not be inferred. It would have been using a full OWL reasoner.

8.5. Using the Transitive Property

If we start by making a few statements about family relationships:

```
(add-triple !kb:relativeOf !rdf:type !owl:TransitiveProperty)
```

```
(add-triple !kb:Mark !kb:relativeOf !kb:Ron)
(add-triple !kb:Ron !kb:relativeOf !kb:Julia)
(add-triple !kb:Julia !kb:relativeOf !kb:Ken)
```

And run a query like:

```
(sparql:run-sparql "
PREFIX kb: <http:://knowledgebooks.com/ontology#>
SELECT ?relative WHERE { kb:Mark kb:relativeOf ?relative }"
 :results-format :sparql-json))
```

If you forget to enable reasoning then you will get results just using RDF + RDFS that you do not expect:

```
{
  "head" : {
    "vars" : ["relative"]
  },
  "results" : {
    "bindings" : [
      {
         "relative":{"type":"uri",
                     "value":
                       "http:://knowledgebooks.com/ontology#Ron"
      }
    ]
  }
}
```

We need to enable reasoning with:

```
(apply-rdfs++-reasoner)
```

and we then get all expected relatives listed:

```
{
  "head" : {
    "vars" : ["relative"]
  },
  "results" : {
    "bindings" : [
```

```
      {
         "relative":{"type":"uri",
                     "value":
                       "http://knowledgebooks.com/ontology#Ron"}
      },
      {
         "relative":{"type":"uri",
                     "value":
                       "http://knowledgebooks.com/ontology#Ken"}
      },
      {
         "relative":{"type":"uri",
                     "value":
                       "http://knowledgebooks.com/ontology#Julia
      }
    ]
  }
}
```

8.6. Wrap Up

We saw in the ”same as” example that RDFS++ does not always make inferences that we might expect, so it is best to test reasoning that you depend on for an application with a small example. At this point, you know how to use AllegroGraph and Lisp to write your ow applications. I am going to take a short tangent in the next chapter to show you a short-hand Prolog notation for using AllegroGraph embedded in Lisp applications.

9. AllegroGraph Prolog Interface

This chapter contains optional material about Franz's proprietary Prolog query interface to AllegroGraph RDF stores. I will not use the Prolog interface in examples later in this book because I wanted to stress standard technologies like SPARQL for accessing RDF data. However, the Prolog interface is very convenient to use and if you will be using AllegroGraph in your work projects I suggest that you take the time to learn it.

The documentation for the Prolog interface[1] is Franz's tutorial and reference web pages. I am going to give you a brief introduction in this chapter and you can later review Franz's documentation if you choose to use the Prolog interface in your projects.

I suggest that you open a Lisp repl and follow along with the examples that are in the file quick_start_allegrograph_lisp_embedded/prolog.lisp. We start by opening a new triple store and loading some example RDF N-Triple data to experiment with:

```
(require :agraph)
(in-package :db.agraph.user)

(enable-!-reader) ; enable the ! reader macro

(create-triple-store "/tmp/rdfstore_prolog_1"
                     :if-exists :supersede)

(register-namespace
 "kb"
 "http://knowledgebooks.com/ontology/#")

(load-ntriples
  #p"quick_start_allegrograph_lisp_embedded/sample_news.nt")
```

I am assuming that you started the Lisp repl in the main examples directory for this book so adjust the path in the **load-ntriples** statement if you started the repl in a different location. I am going to show you a query example and then explain the function of the Prolog operators in the example query:

[1] The Prolog interface is based on Peter Norvig's Prolog implementation written in Common Lisp.

```
(select (?s ?p ?o)
   (q- ?s ?p ?o))
```

This query contains no conditions so every triple is displayed. In Prolog, terms starting with a **?** are variables that will later get value bindings. The **select** Lisp macro is used to perform a query and return results in a convenient Lisp list notation. Each **q-** term in a query is used to define variables and optionally conditions. You will see the close correspondence with SPARQL queries as we look at more examples. The output showing all N-Triples in the example data file looks like:

```
(("http://kbsportal.com/oak_creek_flooding /"
  "http://knowledgebooks.com/ontology/#storyType"
  "http://knowledgebooks.com/ontology/#disaster")
 ("http://kbsportal.com/oak_creek_flooding /"
  "http://knowledgebooks.com/ontology/#summary"
  "Oak Creek flooded last week affecting 5 businesses")
 ("http://kbsportal.com/bear_mountain_fire /"
  "http://knowledgebooks.com/ontology/#storyType"
  "http://knowledgebooks.com/ontology/#disaster")
 ("http://kbsportal.com/bear_mountain_fire /"
  "http://knowledgebooks.com/ontology/#summary"
  "The fire on Bear Mountain was caused by lightening")
 ("http://kbsportal.com/trout_season /"
  "http://knowledgebooks.com/ontology/#storyType"
  "http://knowledgebooks.com/ontology/#sports")
 ("http://kbsportal.com/trout_season /"
  "http://knowledgebooks.com/ontology/#storyType"
  "http://knowledgebooks.com/ontology/#recreation")
 ("http://kbsportal.com/trout_season /"
  "http://knowledgebooks.com/ontology/#summary"
  "Trout fishing season started last weekend")
 ("http://kbsportal.com/jc_basketball /"
  "http://knowledgebooks.com/ontology/#storyType"
  "http://knowledgebooks.com/ontology/#sports"))
```

We can refine this example query by only requesting news stories that have a summary:

```
(select (?news_uri ?summary)
   (q- ?news_uri !kb:summary ?summary))
```

Now the results are:

```
(("http://kbsportal.com/oak_creek_flooding /"
  "Oak Creek flooded last week affecting 5 businesses")
 ("http://kbsportal.com/bear_mountain_fire /"
  "The fire on Bear Mountain was caused by lightening")
 ("http://kbsportal.com/trout_season /"
  "Trout fishing season started last weekend"))
```

If we are only interested in news stories of type disaster, then we can add another condition filtering against the story type:

```
(select (?news_uri ?summary)
   (q- ?news_uri !kb:summary ?summary)
   (q- ?news_uri !kb:storyType !kb:disaster))
```

Now we only get two results:

```
(("http://kbsportal.com/oak_creek_flooding /"
  "Oak Creek flooded last week affecting 5 businesses")
 ("http://kbsportal.com/bear_mountain_fire /"
  "The fire on Bear Mountain was caused by lightening"))
```

The Franz Prolog interface tutorial and reference web pages also show examples of performing RDFS++ type inference and further Prolog techniques. Since we will not use the Prolog interface in application examples in this book I refer you to the Franz documentation if you are interested in using the Prolog interface.

Part III.

Portable Common Lisp Utilities for Information Processing

10. Linked Data and the World Wide Web

It has been a decade since Tim Berners-Lee started writing about “version 2” of the World Wide Web: the Semantic Web. His new idea was to augment HTML anchor links with typed links using RDF data. As we have seen in detail in the last several chapters, RDF is encoded as data triples with the parts of each triple identified as the **subject**, **predicate**, and **object**. The **predicate** identifies the type of link between the **subject** and the **object** in a RDF triple.

You can think of a single RDF graph as being hosted in one web service, SPARQL endpoint service, or a downloadable set of RDF files. Just as the value of the web is greatly increased with relevant links between web pages, the value of RDF graphs is increased when they contain references to triples in other RDF graphs. In theory, you could think of all linked RDF data that is reachable on the web as being a single graph but in practice graphs with billions of nodes are difficult to work with. That said, handling very large graphs is an active area of research both in university labs and in industry.

URIs refer to things, acting as a unique identifier. An important idea is that URIs in linked data sources can also be ”dereferenceable:” a URI can serve as a unique identifier for the Semantic Web and if you follow the link you can find HTML, RDF or any document type that might better inform both human readers and software agents. Typically, a dereferenceable URI is ”followed” by using the HTTP protocol’s GET method.

The idea of linking data resources using RDF extends the web so that both human readers and software agents can use data resources. In Tim Berners-Lee’s 2009 TED talk on Linked Data he discusses the importance of getting governments, companies and individuals to share Linked Data and to not keep it private. He makes the great point that the world has many challenges (medicine, stabilizing the economy, energy efficiency, etc.) that can benefit from unlocked Linked Data sources.

10.1. Linked Data Resources on the Web

There are already many useful public Linked Data sources, with more being developed. Some examples are:

1. DBpedia contains the ”info box” data automatically collected from Wikipedia (see Chapter 14).
2. FOAF (Friend of a Friend) Ontology for specifying information about people and their social and business relationships.
3. GeoNames (http://www.geonames.org/) links place names to DBpedia (see Chapter 15).
4. Freebase (http://freebase.com) is a community driven web portal that allows people to enter facts as structured data. It is possible to query Freebase and get results as RDF. (See Chapter 13).

We have already used the FOAF RDFS definitions in examples in this book[1] and we will DBpedia, GeoNames, and Freebase in later chapters.

10.2. Publishing Linked Data

Leigh Dodds and Ian Davis have written an online book ”Linked Data Patterns”[2] that provides useful patterns for defining and using Linked Data. I recommend their book as a more complete reference than this short chapter.

I have used a few reasonable patterns in this book for defining RDF properties, some examples being:

```
<http://knowledgebooks.com/ontology/containsPlace>
<http://knowledgebooks.com/ontology/containsCity>
<http://knowledgebooks.com/rdf/discusses/person>
<http://knowledgebooks.com/rdf/discusses/place>
```

It is also good practice to name resources automatically using a root URI followed by a unique ID based on the data source; for example: a database row ID or a Freebase ID.

```
<http://knowledgebooks.com/rdf/datasource/freebase/20121>
```

[1] As an example, for people’s names, addresses, etc.
[2] Available under a Creative Commons License at http://patterns.dataincubator.org/book/

```
<http://knowledgebooks.com/rdf/datasource/psql/ \\
testdb/testtable/21198>
```

For all of these examples (properties and resources) it would be good practice to make these URIs dereferenceable.

10.3. Will Linked Data Become the Semantic Web?

There has not been much activity building large systems using Semantic Web technologies. That said, I believe that RDF is a natural data format to use for making statements about data found on the web and I expect the use of RDF data stores to increase. The idea of linked data seems like a natural extension: making URIs dereferenceable lets people follow URIs and get additional information on commonly used RDFS properties and resources. I am interested in Natural Language Processing (NLP) and it seems reasonable to expect that intelligent agents can use natural (human) language dereferenced descriptions of properties and resources.

10.4. Linked Data Wrapup

I have defined the relevant terms for using Linked Data in this short chapter and provided references for further reading and research. Much of the rest of this book is comprised of Linked Data application examples using some utilities for information extraction and processing with existing data sources.

11. Common Lisp Client Library for Open Calais

The Open Calais web services are available for free use with some minor limitations. This service is also available for a fee with additional functionality and guaranteed service levels. We will use the free service in this chapter. Although I made this chapter self-contained, you may also want to read the documentation at **www.opencalais.com**.

You will need to apply for a free developers key. On my development systems I define an environment variable for the value of my key using (the key shown is not a valid key, by the way):

```
export OPEN_CALAIS_KEY=po2eq112hkf985f3k
```

The example source files are found in lisp_practical_semantic_web/opencalais:

- load.lisp – loads and runs the demo
- opencalais-lib.lisp – performs web service calls to find named entities in text
- opencalais-data-store.lisp – maintains an RDF data store for named entities
- test-opencalais.lisp – demo test program

11.1. Open Calais Web Services Client

The Open Calais web services return RDF payloads serialized as XML data that you can print out[1] if you want to see what it looks like.

For our purposes, we will not use the returned XML data and instead parse the comment block to extract named entities that Open Calais indentifies. There is a possibility in the future that the library in this section may need modification if the format of this comment block changes (it has not changed in several years).

[1] Add a line of debug printout for the response returned by the web service call.

I will not list all of the code in opencalais-lib.lisp but we will look at some of it. I start by defining two constant values, the first depends on your setting of the environment variable OPEN_CALAIS_KEY:

```
(defvar *my-opencalais-key* (sys::getenv "OPEN_CALAIS_KEY"))

(defvar *PARAMS*
    (concatenate 'string
      "&paramsXML="
       (MAKE-ESCAPED-STRING
         "<c:params ... >.....</c:params>")))
```

The web services client function is fairly trivial: we just need to make a RESTful web services call and extract the text form the comment block, parsing out the named entities and their values. Before we look at some code, we will jump ahead and look at an example comment block; understanding the input data will make the code easier to follow:

```
<!--Relations: PersonCommunication,
               PersonPolitical,
               PersonTravel

Company: IBM, Pepsi
Country: France
Person: Hiliary Clinton, John Smith
ProvinceOrState: California-->
```

We will use the **net.aserve.client:do-http-request** function to make the web service call after setting up the RESTful arguments:

```
(defun entities-from-opencalais-query (query
              &aux url results index1 index2
              lines tokens hash)
  (setf hash (make-hash-table :test #'equal))
  (setf url
    (concatenate 'string
      "http://api.opencalais.com/enlighten"
      "/calais.asmx/Enlighten?"
      "licenseID="
      *my-opencalais-key*
      "&content="
      (MAKE-ESCAPED-STRING  query)
```

```
      *PARAMS*))
  (setf results (net.aserve.client:do-http-request url))
```

The value of results will be URL-encoded text and for our purposes there is no need to decode the text returned from the web service call:

```
  (setq index1 (search "terms of service.--&gt;" results))
  (setq index1 (search "&lt;!--" results :start2 index1))
  (setq index2 (search "--&gt;" results :start2 index1))
  (setq results (subseq results (+ index1 7) index2))
  (setq lines
    (split-sequence:split-sequence #\Newline results))
  (dolist (line lines)
    (setq index1 (search ": " line))
    (if index1
        (let ((key (subseq line 0 index1))
              (values (split-sequence:split-sequence ", "
                        (subseq line (+ index1 2)))))
          (if (not (string-equal "Relations" key))
              (setf (gethash key hash) values)))))
  (maphash
   #'(lambda (key val)
       (format t "key: ~S val: ~S~%" key val))
   hash)
  hash)
```

Before using this utility function in the next section to fetch data for an RDF data store we will look at a simple test:

```
(entities-from-opencalais-query
    "Senator Hiliary Clinton spoke with the president
     of France.  Clinton and John Smith talked on
     the aiplane going to California. IBM and Pepsi
     contributed to Clinton's campaign.")
```

The debug printout in this call is:

```
key: "Country" val: ("France")
key: "Person" val: ("Hiliary Clinton" "John Smith")
key: "Company" val: ("IBM" "Pepsi")
key: "ProvinceOrState" val: ("California")
```

11.2. Storing Entity Data in an RDF Data Store

We will use the utilities developed in the last section for using the Open Calais web services in this section to populate an RDF data store. You can find the utilities developed in this section in the source file opencalais-data-store.lisp. We start by making sure that the AllegroGraph libraries are loaded and we define a namespace that we will use for examples in the rest of this chapter:

```
;; Use the opencalais-lib.lisp utilities to create
;; an RDF data store. Assume that a AG RDF
;; repository is open.

(require :agraph)
(in-package :db.agraph.user)

(register-namespace
     "kb"
     "http:/knowledgebooks.com/rdfs#")
```

To avoid defining a global variable for a hash table we define one locally inside a closure that also defines the only function that needs read access to this hash table:

```
(let ((hash (make-hash-table :test #'equal)))
  (setf (gethash "Country" hash) !kb:containsCountry)
  (setf (gethash "Person" hash) !kb:containsPerson)
  (setf (gethash "Company" hash) !kb:containsCompany)
  (setf (gethash "ProvinceOrState" hash)
      !kb:containsState)
  (setf (gethash "Product" hash) !kb:containsProduct)
  ;; utility function for getting a URI for a
  ;; predicate name:
  (defun get-rdf-predicate-from-entity-type (entity-type)
    (let ((et (gethash entity-type hash)))
      (if (not et)
          (progn
            ;; just use a string literal if there is
            ;; no entry in the hash table:
            (setf et entity-type)
            (format t
                    "Warning: entity-type ~S not defined
                     in opencalais-data-store.lisp~%"
                    entity-type)))
      et)))
```

Function **get-rdf-predicate-from-entity-type** is used to map string literals to specific predicates defined in the knowledgebooks.com namespace. The following function is the utility for processing the text from documents and generating multiple triples that all have their subject equal to the value of the unique URI for the original document.

```
(defun add-entities-to-rdf-store (subject-uri text)
  "subject-uri if the subject for triples that this
   function defines"
    (maphash
     #'(lambda (key val)
         (dolist (entity-val val)
           (add-triple
            subject-uri
            (get-rdf-predicate-from-entity-type key)
            (literal entity-val))))
     (entities-from-opencalais-query text)))
```

If documents are not plain text (for example a word processing file or a HTML web page) then applications using the utility code developed in this chapter need to extract plain text. The code in this section is intended to give you ideas for your own applications; you would at least substitute your own namespace(s) for your application.

11.3. Testing the Open Calais Demo System

The source file test-opencalais.lisp contains the examples for this section:

```
(require :agraph)
(in-package :db.agraph.user)

(create-triple-store "/tmp/rdfstore_1")
```

We start by using the utility function defined in the last section to find all named entities in sample text and create triples in the data store:

```
(add-entities-to-rdf-store
     !<http://newsdemo.com/1234>
     "Senator Hiliary Clinton spoke with the president
      of France. Clinton and John Smith talked on the
      aiplane going to California. IBM and Pepsi
      contributed to Clinton's campaign.")
```

We can print all triples in the data store:

```
(print-triples (get-triples-list) :format :concise)
```

and output is:

```
<1: http://newsdemo.com/1234 kb:containsCountry France>
<2: http://newsdemo.com/1234 kb:containsPerson
                             Hiliary Clinton>
<3: http://newsdemo.com/1234 kb:containsPerson
                             John Smith>
<4: http://newsdemo.com/1234 kb:containsCompany IBM>
<5: http://newsdemo.com/1234 kb:containsCompany Pepsi>
<6: http://newsdemo.com/1234 kb:containsState California>
```

This example showed just adding triples generated from a single document. If a large number of documents are processed then queries like the following might be useful:

```
(print-triples
 (get-triples-list
  :p (get-rdf-predicate-from-entity-type "Company")
  :o (literal "Pepsi"))
 :format :concise)
```

producing this output:

```
<5: http://newsdemo.com/1234 kb:containsCompany Pepsi>
```

Here we identify all documents that mention a specific company.

11.4. Open Calais Wrap Up

Since AllegroGraph supports indexing and search of any text fields in triples, the combination of using triples to store specific entities in a large document collection with full search, AllegroGraph can be a effective tool to mange large document repositories.

“Documents” can be any source of text identified with a unique URI: web pages, word processing documents, blog entries, etc.

I consider Open Calais to be state-of-the-art in its ability to accurately determine entities in input text. In the next chapter I will show you my own Common Lisp library that I use when I do not want to depend on accessing a third party web service like Open Calais.

12. Common Lisp Client Library for Natural Language Processing

We used the Open Calais web services in Chapter 11 to identify proper names in text and we used this information to create RDF data linking information sources with people and places mentioned in the text. I will introduce you to a subset of my own library for natural language processing (NLP) in this chapter.

12.1. KnowledgeBooks.com Natural Language Processing Library

I started developing the KnowledgeBooks.com NLP library as a commercial product in the 1990s and continued to improve it until about 2008 when I started using the Open Calais web services for most of my own work and research. While the Open Calais system produces more accurate results than my own library, I still find it useful to use my own library so I took some effort to clean up my old code (actually removing most of it, leaving the parts that you may find most useful and easiest to understand.) for inclusion in the software distribution for this book.

The code I discuss in this chapter is a subset of my library. I wanted to include only the functionality to replace Open Calais.

I will give you a quick tutorial in this chapter on using the major APIs in my library. If you are interested in the code itself then you can browse through the code. I set up my library to load using the ASDF package manager. If you are running a Lisp repl from the knowledgebooks_nlp subdirectory then you can load the library directly using:

```
(asdf:operate 'asdf:load-op :kbnlp)
(in-package :kbnlp)
```

If you are working in the top level examples directory for this book then load my library using:

```
(push "knowledgebooks_nlp/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op :kbnlp)
(in-package :kbnlp)
```

If you want to use my library from a different directory location you will need to push the path of the knowledgebooks_nlp directory to **asdf:*central-registry***.[1]

The file knowledgebooks_nlp/example.lisp contains sample code for using the library:

```
(defvar x
  (kbnlp:make-text-object "President ..."))
```

A **text-object** Lisp struct contains attributes for a summary of the text, human name and place name entities found in the text, part of speech tags, and topic tags describing the input text:

```
(defvar x
  (kbnlp:make-text-object "President ..."))
```

With most of the output not shown for brevity, here is the output after loading the file example.lisp:

```
#S(TEXT :URL "http://knowledgebooks.com/docs/001"
        :TITLE "test doc 1"
        :SUMMARY "Often those amendments are ..."
        :CATEGORY-TAGS (("news_politics.txt" 0.38268)
                        ("news_economy.txt" 0.31182)
                        ("news_war.txt" 0.20174))
        :HUMAN-NAMES ("President Bill Clinton")
        :PLACE-NAMES ("Florida")
        :TEXT #("President" "Bill" "Clinton" "ran"  ...)
        :TAGS #("NNP" "NNP" "NNP" "VBD" "IN" "NN" ...))
```

The part of speech tags are defined in my FastTag project that you can download from my Open Source web page makrwatson.com/opensource.

[1] The data file loading code depends on the exact directory name knowledgebooks_nlp so do not change it.

12.2. KnowledgeBooks Natural Language Processing Library Wrapup

I will be using my library in Chapter 16 to process input text sources and create RDF triples containing semantic data for input text. Chapter 16 contains example code for effectively using my library in larger systems written in Lisp.

There is some initialization time overhead using my library: on my laptop it takes about five seconds to load all of the linguistic data files[2] included with the book example code but after this initial setup my library is very fast.[3]

[2] I provide small linguistic example data files with the example code for this book that should be sufficient for experimenting with my library and use in most applications. For my own research and work for my consulting customers I use a large linguistic data set that takes about thirty seconds to load.

[3] It takes about 3 milliseconds to process 100 words of input text on my MacBook.

13. Common Lisp Client Library for Freebase

Freebase is a public data source created by the MetaWeb Corporation. Freebase is similar to Wikipedia because users of Freebase add data that is linked to other data in Freebase. If you are not already familiar with Freebase then I suggest you spend some time experimenting with the web interface (http://freebase.com) before working through this chapter. As a developer make sure that you eventually look at the developer documentation at http://www.freebase.com/docs/data because I will only the aspects of Freebase that I need for the example applications in this book.

13.1. Overview of Freebase

Objects stored in Freebase have a unique object ID assigned to them. It makes sense to use this ID as part of a URI when generating URIs to use as RDF resources. We talked about dereferenceable URIs in Section 3.3. The RDF for the object representing me on Freebase can be obtained by dereferencing:

```
http://rdf.freebase.com/rdf/ \\
guid.9202a8c04000641f80000000146fb902
```

Objects in Freebase are tagged with one or more types. For example, if I search for myself and fetching HTML output using a URI like:

```
http://www.freebase.com/search?query=Mark+Watson+consultant
```

then I see that I am assigned to three types: Person, Author, and Software Developer. If I want JSON formatted results then I can use:

```
http://www.freebase.com/api/service/search?query= \\
Mark+Watson+author
```

A full reference of API arguments is http://www.freebase.com/view/en/api_service_search and Table 13.1 shows the arguments that I most frequently use.

Table 13.1.: Subset of Freebase API Arguments

Argument	Argument type	Format	Default value
query	required	string	
type	optional	string	/location/citytown
limit	optional	integer	20
start	optional	integer	0

The best way to access Freebase is to use the MQL Query Language. However, Freebase now has an RDF interface[1] and we will use this interface in this chapter.

Please note that the Java, Clojure, JRuby, and Scala edition of this book wraps the Java Freebase MQL client library for full access to Freebase. You can refer to the other edition of this book for more detailed information concerning Freebase.

http://rdf.freebase.com/

The RDF interface can fetch all RDF triples for a given Freebase RDF resource identifier. As an example, here is the identifier for the Freebase topic about me:

```
http://rdf.freebase.com/ns/en.mark_louis_watson
```

The returned triples are (most not shown for brevity):[2]

```
<http://rdf.freebase.com/ns/en.mark_louis_watson>
  <http://rdf.freebase.com/ns/people.person.date_of_birth>
  "1951" .
<http://rdf.freebase.com/ns/en.mark_louis_watson>
  <http://rdf.freebase.com/ns/common.topic.alias>
  "Mark Watson"@en .
<http://rdf.freebase.com/ns/en.mark_louis_watson>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://rdf.freebase.com/ns/computer.software_developer> .
<http://rdf.freebase.com/ns/en.mark_louis_watson>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://rdf.freebase.com/ns/book.author> .
<http://rdf.freebase.com/ns/en.mark_louis_watson>
  <http://creativecommons.org/ns#attributionName>
  "Source: Freebase - The World's database" .
```

[1] http://blog.freebase.com/2008/10/30/introducing_the_rdf_service/
[2] Output edited to fit page width.

```
<http://rdf.freebase.com/ns/en.mark_louis_watson>
  <http://rdf.freebase.com/ns/people. \\
              person.education>
  <http://rdf.freebase.com/ns/m.0b6_ggq> .
<http://rdf.freebase.com/ns/m.0b6_ggq>
  <http://rdf.freebase.com/ns/education. \\
            education.institution>
  <http://rdf.freebase.com/ns/en.university_of_california_santa_b
```

You can use the Freebase RDF browser[3] to find Freebase RDF resource identifiers using free text search.

13.2. Accessing Freebase from Common Lisp

I created a short file freebase_client/test.lisp that shows you how to create an MQL query, encode it as JSON, and make a web service call to Freebase. For this example I want to create JSON data that looks like:

```
[{
  "name":   "Mark Louis Watson",
  "type": []
}]
```

The file test.lisp creates this JSON query request and makes a web service call:

```
(require :aserve)
(in-package :net.aserve.client)

(push "../utils/yason/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op 'yason)

(defvar mql-url
    "http://api.freebase.com/api/service/mqlread?query=")

(defvar *h* (make-hash-table :test #'equal))
(defvar *h2* (make-hash-table :test #'equal))
(setf (gethash "name" *h2*) "Mark Louis Watson")
(setf (gethash "type" *h2*) (make-array 0))
(setf (gethash "query" *h*) (list *h2*))
```

[3] http://rdf.freebase.com/

```
(defvar *hs*
  (with-output-to-string
    (sstrm)
    (json:encode *h* sstrm)))

(defvar *s*
  (concatenate 'string
      mql-url
      (net.aserve.client::uriencode-string *hs*)))

(defvar *str-results* (do-http-request *s*))

(format t "Results:~%~%~A~%~%" *str-results*)
```

The output is a string containing encoded JSON data and looks like:

```
{
  "code": "/api/status/ok",
  "result": [
    {
      "name": "Mark Louis Watson",
      "type": [
        "/common/topic",
        "/people/person",
        "/book/author",
        "/computer/software_developer"
      ]
    }
  ],
  "status": "200 OK",
  "transaction_id": "cache;cache04.p01;2010-10-23T22"
}
```

This code snippet gets the result as Lisp data:

```
(defvar *results* (json:parse *str-results*))

(maphash
  #'(lambda (key val)
      (format t "key: ~A value: ~A~%" key val))
  (car (gethash "result" *results*)))
```

The output looks like:

```
key: name value: Mark Louis Watson
key: type value: (/common/topic /people/person
                  /book/author
                  /computer/software_developer)
```

13.3. Freebase Wrapup

Freebase is a useful source of semantic data and this chapter introduced you to accessing Freebase in general and from Lisp client code. One issue with Freebase is that it contains sparse data: some topics are well covered and others are not. If you use Freebase in your Lisp applications start with the interactive query editor[4] to explore the available data and get valid MQL queries for the information you want. Once you have valid MQL queries then use the Lisp code example from the last section to convert your MQL queries to JSON data and call the Freebase web services.

[4] http://www.freebase.com/app/queryeditor

14. Common Lisp Client Library for DBpedia

This Chapter will cover the development of a general purpose SPARQL client library and also the use of this library to access the DBpedia SPARQL endpoint.

DBpedia is a mostly automatic extraction of RDF data from Wikipedia using the metadata in Wikipedia articles. You have two alternatives for using DBpedia in your own applications: using the public DBpedia SPARQL endpoint web service or downloading all or part of the DBpedia RDF data and loading it into your own RDF data store (e.g., AllegroGraph or Sesame).

The public DBpedia SPARQL endpoint URI is http://dbpedia.org/sparql. For the purpose of the examples in this book we will simply use the public SPARQL endpoint but for serious applications I suggest that you run your own endpoint using the subset of DBpedia data that you need..

The public DBpedia SPARQL endpoint is run using the Virtuoso Universal Server (http://www.openlinksw.com/). If you want to run your own your own DBpedia SPARQL endpoint you can download the RDF data files from http://wiki.dbpedia.org and use the open source version of Virtuoso, Sesame, AllegroGraph, or any other RDF data store that supports SPARQL queries.

14.1. Interactively Querying DBpedia Using the Snorql Web Interface

When you start using DBpedia, a good starting point is the interactive web application that accepts SPARQL queries and returns results. The URL of this service is:

```
http://dbpedia.org/snorql
```

Figure 14.1 shows the DBpedia Snorql web interface showing the results of one of the sample SPARQL queries used in this section.

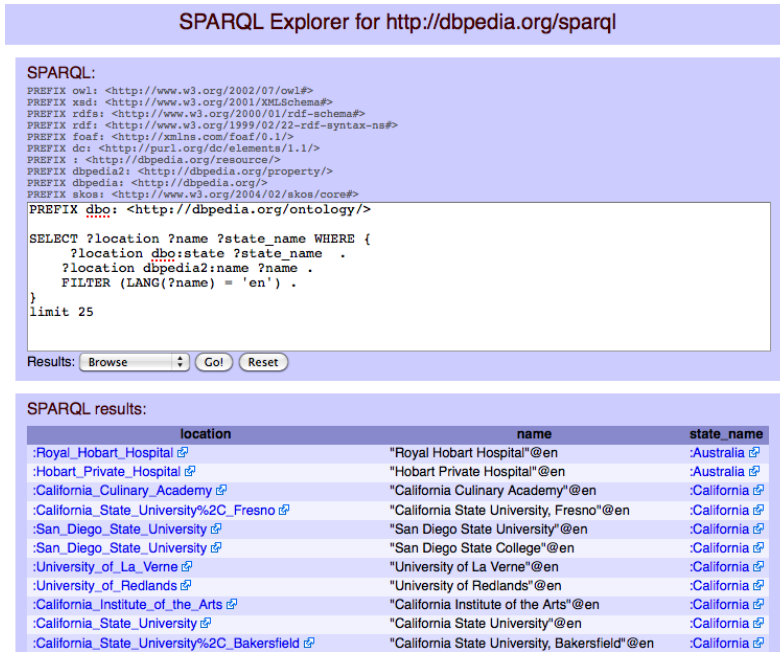

Figure 14.1.: DBpedia Snorql Web Interface

A good way to become familiar with the DBpedia ontologies used in these examples is to click the links for property names and resources returned as SPARQL query results, as seen in Figure 14.1. Here are three different sample queries that you can try:

```
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?s ?p WHERE {
      ?s ?p <http://dbpedia.org/resource/Berlin> .
}
ORDER BY ?name
```

```
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?s ?p WHERE {
      ?s dbo:state ?p  .
}
limit 25
```

```
PREFIX dbpedia2: <http://dbpedia.org/property/>
```

```
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?location ?name ?state_name WHERE {
       ?location dbo:state ?state_name  .
       ?location dbpedia2:name ?name .
       FILTER (LANG(?name) = 'en') .
}
limit 25
```

The http://dbpedia.org/snorql SPARQL endpoint web application is a great resource for interactively exploring the DBpedia RDF datastore. We will look at an alternative browser in the next section.

14.2. Interactively Finding Useful DBpedia Resources Using the gFacet Browser

The gFacet browser allows you to find RDF resources in DBpedia using a search engine. After finding matching resources you can then dig down by clicking on individual search results.

You can access the gFacet browser using this URL:

```
http://www.gfacet.org/dbpedia/
```

Figures 14.2 and 14.3 show a search example where I started by searching for "Arizona parks," found five matching resources, clicked the first match "Parks in Arizona," and then selected "Dead Horse State Park."[1]

14.3. The lookup.dbpedia.org Web Service

We will use Georgi Kobilarov's DBpedia lookup web service to perform free text search queries to find data in DBpedia using free text search. If you have a good idea of what you are searching for and know the commonly used DBpedia RDF properties then using the SPARQL endpoint is convenient. However, it is often simpler to just perform a keyword search and this is what we will use the lookup web service for. We will later see the implementation of a client library in Section **??**. You can find documentation on the REST API at http://lookup.dbpedia.org/api/search.asmx?op=KeywordSearch. Here is an example URL for a REST query:

[1] This is a park near my home where I go kayaking and fishing.

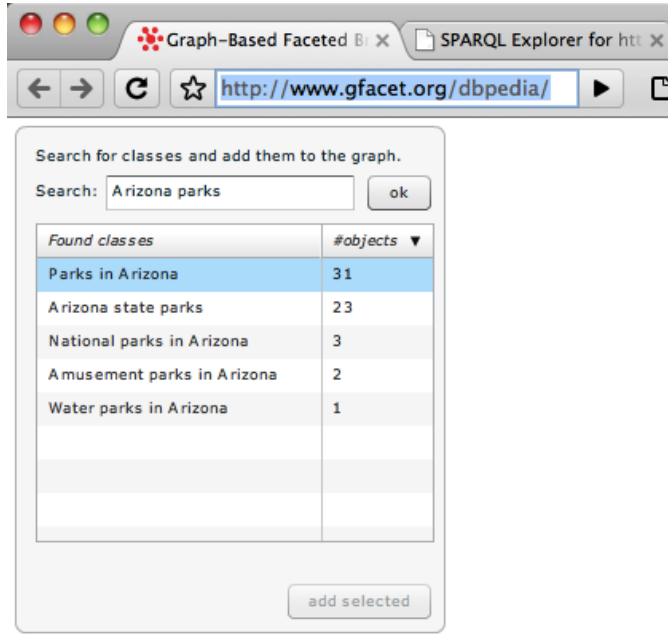

Figure 14.2.: DBpedia Graph Facet Viewer

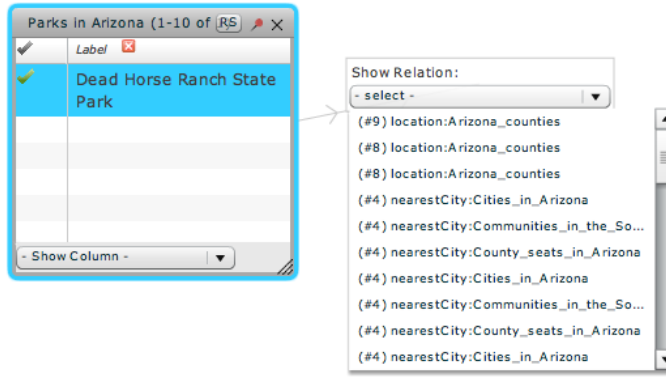

Figure 14.3.: DBpedia Graph Facet Viewer after selecting a resource

```
http://lookup.dbpedia.org/api/search.asmx/KeywordSearch? \\
QueryString=Flagstaff\&QueryClass=XML\&MaxHits=10
```

As you will see in Section **??**, the search client needs to filter results returned from the lookup web service since the lookup service returns results with partial matches of search terms. I prefer to get only results that contain all search terms.

The following sections contain implementations of a SPARQL client and a free text search lookup client.

DBpedia is a mostly automatic extraction of RDF data from Wikipedia using the metadata in Wikipedia articles. You have two alternatives for using DBpedia in your own applications: using the public DBpedia SPARQL endpoint or downloading all or part of the DBpedia RDF data and loading it into your own RDF data store (e.g., AllegroGraph or Sesame).

The public DBpedia SPARQL endpoint URI is http://dbpedia.org/sparql. For the purpose of the examples in this book we will simply use the public SPARQL endpoint but for serious applications I suggest that you run your own endpoint.

14.4. Using the AllegroGraph SPARQL Client Library to access DBpedia

The AllegroGraph SPARQL Client library makes it very simple to use the public DBPedia web service. I have an example in dbpedia/test.lisp that shows how to run a sample query:[2]

```
markws-macbook:lisp_practical_semantic_web markw$ cd dbpedia/
markws-macbook:dbpedia markw$ lisp
CL-USER(1): :ld test
CL-USER(2): (sparql.client::run-sparql-remote
    "http://dbpedia.org/sparql" "
 PREFIX dbpedia: <http://dbpedia.org/ontology/>
 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
 SELECT ?person {
   ?person
   dbpedia:birthPlace
   <http://dbpedia.org/resource/Boston> .
 }
 LIMIT 6" :results-format :alists)
```

[2] I had to edit this output to fit the page width.

```
((((|?person| .
      !<http://dbpedia.org/resource/Benjamin_Franklin>))
 ((|?person| .
      !<http://dbpedia.org/resource/Cotton_Mather>))
 ((|?person| .
      !<http://dbpedia.org/resource/James_Spader>))
 ((|?person| .
      !<http://dbpedia.org/resource/Christa_McAuliffe>))
 ((|?person| .
      !<http://dbpedia.org/resource/Gordon_K_MacLeod>))
 ((|?person| .
      !<http://dbpedia.org/resource/Edgar_Allan_Poe>)))
:SELECT
#(|?person|)
```

14.5. DBpedia Wrapup

DBPedia is a great information resource and the AllegroGraph SPARQL client library makes it easy to make queries and use the results in Lisp applications. I will not use DBPedia in any further examples in this book but I wanted to show you how to use DBpedia in your Lisp applications.

15. Library for GeoNames

GeoNames (http://www.geonames.org/) is a geographic information database. The raw data is available under a Creative Commons Attribution license. There is a free web service and a commercial web service. For production environments you will want to use the commercial service but for development purposes and for the examples for this book I use the free service[1].

15.1. Using the cl-geonames Library

We will use the Common Lisp GeoNames client by Nicolas Lamirault[2] in this chapter. I installed cl-geonames and all dependencies[3] in the utils directory for the examples in in this book.

The file in geonames/test.lisp contains an example of loading and running a few cl-geonames examples:

```
mark:lisp_practical_semantic_web markw$ cd geonames/
mark:geonames markw$ alisp
CL-USER(1): :ld test
CL-USER(2): (cl-geonames:geo-country-info :country '("US" "FR"))
(:|geonames|
 (:|country| (:|countryCode| "FR") (:|countryName| "France")
  (:|isoNumeric| "250") (:|isoAlpha3| "FRA") (:|fipsCode| "FR")
  (:|continent| "EU") (:|capital| "Paris") (:|areaInSqKm| "547030
  (:|population| "64768389") ...)
 (:|country| (:|countryCode| "US") (:|countryName| "United States
  (:|isoNumeric| "840") (:|isoAlpha3| "USA") (:|fipsCode| "US")
  (:|continent| "NA") (:|capital| "Washington")
  (:|areaInSqKm| "9629091.0") (:|population| "310232863") ...))
```

[1] The geonames.org web service is limited to 2000 queries per hour from any single IP address. Commercial support is available, or, with some effort, you can also run GeoNames on your own server with some effort. There are, for example, a few open source Ruby on Rails projects that use the Geonames data files and provide a web service interface.

[2] http://code.google.com/p/cl-geonames/

[3] Drakma, s-xml, cl-json, chunga, cl-base64, flexi-streams, puri, split-sequence, trivial-gray-streams, usocket

```
CL-USER(3): (cl-geonames::geo-country-code "42.21" "-71.5")
(:|geonames|
 (:|country| (:|countryCode| "US") (:|countryName| "United State
  (:|distance| "0.0")))
CL-USER(4): (cl-geonames::geo-elevation-srtm3 "42.21" "-71.5")
"122"
CL-USER(5): (cl-geonames::geo-country-subdivision "42.21" "-71.5
(:|geonames|
 (:|countrySubdivision| (:|countryCode| "US")
  (:|countryName| "United States") (:|adminCode1| "MA")
  (:|adminName1| "Massachusetts") ((:|code| :|type| "FIPS10-4")
  ((:|code| :|type| "ISO3166-2") "MA") (:|distance| "0.0")))
CL-USER(6): (cl-geonames::geo-find-nearby-place-name "42.21" "-7
(:|geonames|
 (:|geoname| (:|toponymName| "Hayden Row") (:|name| "Hayden Row"
  (:|lat| "42.20426") (:|lng| "-71.51062") (:|geonameId| "493915
  (:|countryCode| "US") (:|countryName| "United States") (:|fcl|
  (:|fcode| "PPL") ...)
 (:|geoname| (:|toponymName| "Hopkinton") (:|name| "Hopkinton")
  (:|lat| "42.22358") (:|lng| "-71.52282") (:|geonameId| "725769
  (:|countryCode| "US") (:|countryName| "United States") (:|fcl|
  (:|fcode| "PPL") ...)
 (:|geoname| (:|toponymName| "Hopkinton") (:|name| "Hopkinton")
  (:|lat| "42.22871") (:|lng| "-71.52256") (:|geonameId| "493988
  (:|countryCode| "US") (:|countryName| "United States") (:|fcl|
  (:|fcode| "PPL") ...)
 (:|geoname| (:|toponymName| "Camp Bob White")
  (:|name| "Camp Bob White") (:|lat| "42.22648") (:|lng| "-71.46
  (:|geonameId| "4932024") (:|countryCode| "US")
  (:|countryName| "United States") (:|fcl| "P") (:|fcode| "PPL")
 (:|geoname| (:|toponymName| "North Milford") (:|name| "North Mi
  (:|lat| "42.18343") (:|lng| "-71.53784") (:|geonameId| "494567
  (:|countryCode| "US") (:|countryName| "United States") (:|fcl|
  (:|fcode| "PPL") ...))
CL-USER(7): (cl-geonames::geo-search "Sedona" "Sedona" "Sedona"
((:|geonames| :|style| "MEDIUM") (:|totalResultsCount| "1")
 (:|geoname| (:|toponymName| "Sedona") (:|name| "Sedona")
  (:|lat| "34.86974") (:|lng| "-111.76099") (:|geonameId| "53136
  (:|countryCode| "US") (:|countryName| "United States") (:|fcl|
  (:|fcode| "PPL")))
```

15.2. Geonames Wrapup

We will not use Geonames in any further examples in this book but I wanted to show you how to load and use cl-geonames since it is a great resource when you are dealing with data for countries, cities, etc. The Freebase database also has geographic data.

Part IV.

Example Semantic Web Application

16. Semantic Web Portal Back End Services

The web portal application developed in this chapter and in Chapter 17 is meant to show you several useful techniques: how to use an RDF data store instead of a relational database, how to organize complex information as RDF data, and how to write a high performance web application using Common Lisp and the open source Portable Allegroserve library[1].

In this chapter I will first list all of the ”back end” functionality that the web UI developed in Chapter 17 will need. Then we will implement this functionality in a single file web_app/backend.lisp. The required functionality is:

1. Read initial RDF data from a file init.nt that contains a few login accounts and the port number that the web application will use.
2. Check for valid user login.
3. Utilities for entering new ”documents” into the system: perform NLP semantic analysis, save semantic tags, entities, input text in AllegroServe.
4. Search wrapper: given search terms, return matching document IDs.
5. Given a document ID, return all information about the document.

I am going to use my own KnowledgeBooks NLP library in this chapter but a good exercise for you would be to make an alternative version that uses the client library for Open Calais that I wrote for Chapter 11.

After backend.lisp is written then it will be fairly easy to write the web application UI in Chapter 17.

[1] The part of this example using AllegroGraph is specific to Franz Lisp and AllegroGraph but what you will learn in Chapter 17 will work well with other Common Lisp implementations like SBCL.

16.1. Implementing the Back End APIs

This pedantic example web application substitutes the use of AllegroGraph instead of a relational database for all data storage requirements. A real application would probably use a relational database to store user information and AllegroGraph to store semantic data.

The file web_app/backend.lisp contains the implementation of the back end APIs and you should open this file in a text editor while you read through this section because I will only show you as few code snippets in the book text. I start by loading my NLP library and the AllegroGraph library and performing some AllegroGraph initialization as seen in earlier book examples:

```
(push "../knowledgebooks_nlp/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op :kbnlp)

(eval-when (compile load eval)
  (require :aserve)
  (require :agraph))
```

Here I loaded my NLP library, the open source portable AllegroServe library, and the embedded AllegroGraph library. The following code snippet performs the same AllegroGraph setup that we have already seen and loads the application parameters from an N-Triple RDF file into our local RDF data store using the file path **/tmp/web-portal_rdf**:

```
(defpackage :user (:use :net.aserve.client :kbnlp))
(in-package :user)

(db.agraph.user::enable-!-reader)

(db.agraph.user::create-triple-store
   "/tmp/webportal\_rdf")
(db.agraph.user::register-namespace
  "kb" "http://knowledgebooks.com/rdfs#")
(db.agraph.user::register-freetext-predicate
   !kb:docTitle)
(db.agraph.user::register-freetext-predicate
   !kb:docText)

(db.agraph.user::load-ntriples #p"init.nt")
```

The most interesting code in backend.lisp is the function for adding a new document:

```
(defun add-document (doc-uri doc-title doc-text)
  (let* ((txt-obj
          (kbnlp:make-text-object doc-text
                                  :title doc-title
                                  :url doc-uri))
         (resource (db.agraph.user::resource doc-uri)))
    (db.agraph.user::add-triple resource
                                !rdf:type !kb:document)
    (db.agraph.user::add-triple resource
                                !kb:docTitle
                                (db.agraph.user::literal
                                   doc-title))
    (db.agraph.user::add-triple resource
                                !kb:docText
                                (db.agraph.user::literal
                                   doc-text))
    (dolist (human-name (kbnlp::text-human-names txt-obj))
      (pprint human-name)
      (db.agraph.user::add-triple resource
                                  !kb:docPersonEntity
                                  (db.agraph.user::literal
                                     human-name)))
    (dolist (place-name (kbnlp::text-place-names txt-obj))
      (pprint place-name)
      (db.agraph.user::add-triple resource
                                  !kb:docPlaceEntity
                                  (db.agraph.user::literal
                                     place-name)))
    (dolist (tag (kbnlp::text-category-tags txt-obj))
      (pprint tag)
      (db.agraph.user::add-triple
        resource !kb:docTag
        (db.agraph.user::literal
           (format nil "~A/~A" (car tag) (cadr tag)))))))
```

Here I used my NLP library but as an exercise, you could rewrite this using the Open Calais client library I provided in Chapter 11. It can be useful having a local entity extraction library so applications do not need access to the Internet.[2]

The function **doc-search** is a simple wrapper for using the AllegroGraph text search APIs:

[2] As I write this chapter in October 2010, I am on a ship in the Pacific Ocean with a poor Internet connection.

```
(defun doc-search (search-term-string)
  "return a list of matching doc IDs"
  (db.agraph.user::freetext-get-ids search-term-string))
```

Search results are retuned as a list of document IDs and the following function **get-doc-info** can be used to reconstruct a document object from the RDF triples containing original data and semantic data for a document:

```
(defun get-doc-info (doc-id)
  (let* ((parts
            (db.agraph.user::get-triple-by-id doc-id))
         (subject (db.agraph.user::subject parts))
         (predicate (db.agraph.user::predicate parts))
         (object (db.agraph.user::object parts)))
    (mapcar
     #'(lambda (obj)
         (list
           (db.agraph.user::part->concise
              (db.agraph.user::subject obj))
           (db.agraph.user::part->concise
              (db.agraph.user::predicate obj))
           (db.agraph.user::part->terse
              (db.agraph.user::object obj))))
      (db.agraph.user::get-triples-list :s subject))))
```

I don't list them here, but the file **backend.lisp** also contains utility functions **print-all-docs** and **delete-all-docs** that you might find useful if you interactively experiment with the code in this chapter.

16.2. Unit Testing the Backend Code

I provide a few unit tests in the file **utils/lisp-unit.lisp** to test document creation and storage in AllegrGraph and test login functionality:

```
(load "../utils/lisp-unit.lisp")
(use-package :lisp-unit)
(load "backend.lisp")

(lisp-unit:define-test "create-doc-test1"
  (add-document "file:///test1.doc" "test title"
                "John Smith went to Mexico")
```

```
  (let ((person-list (db.agraph.user::get-triples-list
                      :p !kb:docPersonEntity))
        (place-list (db.agraph.user::get-triples-list
                     :p !kb:docPlaceEntity)))
    (lisp-unit:assert-equal
     (db.agraph.user::part->string
      (db.agraph.user::object (car person-list)))
     "\"John Smith\"")
    (lisp-unit:assert-equal
     (db.agraph.user::part->string
      (db.agraph.user::object (car place-list)))
     "\"Mexico\"")))

(lisp-unit:define-test "print-triples"
  (print-all-docs)
  (lisp-unit:assert-equal t t))

(lisp-unit:define-test "good-login"
  (lisp-unit:assert-equal t
                          (valid-login? "demo" "demo")))

(lisp-unit:define-test "bad-login"
  (lisp-unit:assert-equal nil
                          (valid-login? "demo" "demo2")))

(run-tests)

(db.agraph.user::delete-triples :o !kb:document)
(db.agraph.user::delete-triples :p !kb:docText)
(db.agraph.user::delete-triples :p !kb:docTitle)
(db.agraph.user::delete-triples :p !kb:docPersonEntity)
(db.agraph.user::delete-triples :p !kb:docPlaceEntity)
(db.agraph.user::delete-triples :p !kb:docTag)
(db.agraph.user::delete-triples :p !kb:doc)
```

The following output shows the test results:

```
bad-login: 1 assertions passed, 0 failed.
create-doc-test1: 2 assertions passed, 0 failed.
checking login: demo demo
good-login: 1 assertions passed, 0 failed.
print-triples: 1 assertions passed, 0 failed.
TOTAL: 5 assertions passed, 0 failed, 0 execution errors.
```

.

16.3. Backend Wrapup

I hope that this short chapter served as a good introduction to using Common Lisp to build web applications and for using AllegroGraph. When I build web applications[3] I like to start by implementing and testing the ”back end” server side processing as I have done in this chapter. In the next chapter I develop the ”front end” for a simple web application that builds on the code in this chapter.

[3] In Java, Ruby on Rails, or in Lisp

17. Semantic Web Portal User Interface

I implemented the backend functionality for a semantic web information portal in Chapter 16 and I will use the open source Portable AllegroServe library for developing web applications in Common Lisp and also the open source Dojo Javascript library in this chapter to write a web interface.

I have used the combination of AllegroServe and Dojo on two customer projects in the last four years and it is a ”comfortable” development environment.[1]

17.1. Portable AllegroServe

Portable AllegroServe is an open source project that runs under most Common Lisp implementations. It is bundled with Franz Lisp so you can just require it:

```
(require :aserve)
(require :webactions)
```

The second library loaded in tis code snippet is WebActions that is an open source framework for handling user sessions and other web application ”boilerplate.” You can find full documentation here **opensource.franz.com/aserve/webactions.html**. Please read through the introduction to the WebActions documentation on the web. WebActions uses something similar to Java Server Pages (JSP): Common Lisp Server Pages (CLP). CLP files are HTML with special tags for calling out to Lisp code.

17.2. Layout of CLP files for Web Application

You can find the CLP files for the example application in the directory **web_app/web**:

1. about.clp - an about web page for this application

[1] However, Ruby on Rails is still my preferred web application framework.

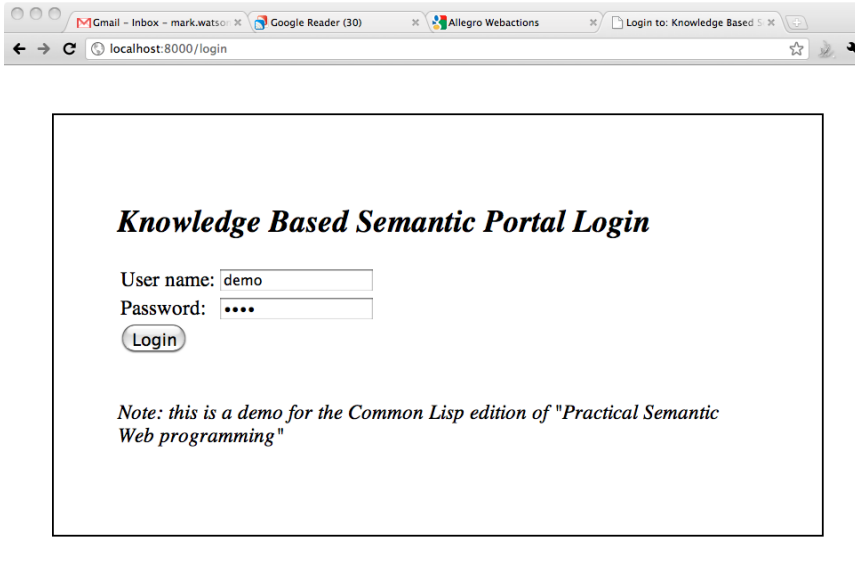

Figure 17.1.: Example Semantic Web Application Login Page

2. browser.clp - this is the page for supporting search and content browsing
3. footer.clp - this is a page fragment that is included at the bottom of most web pages for this application
4. header.clp - this is a page fragment that is included at the top of most web pages for this application
5. left_bar.clp - this page fragment implements the left hand side of the page navigation menu
6. login.clp - contains fields for a user to enter account name and password
7. upload.clp - contains an HTML form for specifying a local file to upload to the web application

Figure 17.1 shows the login page.

17.3. Common Lisp Code for Web Application

We listed the CLP template files in the last section and hopefully you have looked at the online documentation. These template files contained some embedded tags for calling out to Lisp code that is found in the file webapp.lisp. This Lisp source file

also contains all other code required for the web application. I am going to assume that you open the file webapp.lisp in your favorite text editor and read through it; here I will only list and discuss bits of code that are either non-intuitive or especially interesting. I start by requiring libraries and loading my own code:[2][3]

```
(eval-when (compile load eval)
  (require :aserve)
  (require :webactions)
  (load "backend.lisp")
  (load "../utils/file-utils.lisp"))

(defpackage :user (:use :net.aserve :net.html.generator))
(in-package :user)
```

The following code snippet registers WebActions controller functions with HTTP GTE and POST actions:

```
(webaction-project "dojotest"
   :destination "web/"
   :index "login"
   :map
   '(("menu"   action-check-login)
     ("menu"    "menu.clp")
     ("browser" "browser.clp")
     ("search" "/do-search")
     ("admin" "/upload.clp")
     ("about" "/about.clp")
     ("wiki" "/wiki.clp")
     ("upload" "/upload.clp")
     ;;("upload" "/do-upload" "/do-upload" (:redirect t))
     ("gotlogin" action-got-login)
     ("login"  "login.clp")))
```

The login.clp file contains an HTML form for the user's account and password:[4]

```
<form id="myForm2" action="gotlogin" method="post">
  User name:
    <input type="text" name="username" value="demo" />
```

[2] I am loading the source files for my own utilities so they will not be natively compiled when using Franz Lisp. Other Common Lisp implementations like SBCL will compile code that is loaded from source.

[3] Please note that the code snippets shown in this section are not in the same order as they appear in the source file

[4] Formatting HTML is not shown.

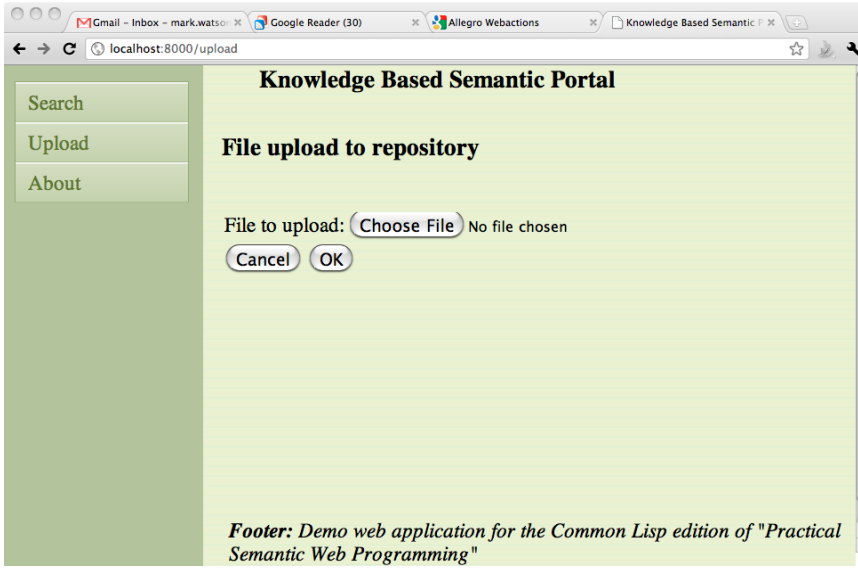

Figure 17.2.: Example File Upload Page

```
  Password:
    <input type="password" name="password" value="demo"/>
</form>
```

The action **gotlogin** checks to see if there is user data in the current request:

```
(defun action-got-login (req ent)
  (let ((session (websession-from-req req)))
    (let ((user  (cdr (assoc "username"
                             (net.aserve:request-query req)
                             :test #'equal)))
          (passwd (cdr (assoc "password"
                             (net.aserve:request-query req)
                             :test #'equal))))
      (if* (and user passwd (valid-login? user passwd))
           then ; already logged in
           "browser"  ; just go to the real home
           else ; must login
           "login"))))
```

Figure 17.2 shows the login page.

The WebActions controller function that handles search requests on the browser page uses the utilities in the file backend.lisp that I discussed in Chapter 16:

```
(defun do-search (req ent)
  (net.aserve:with-http-response (req ent)
    (net.aserve:with-http-body (req ent)
      (let ((test-input
              (cdr (assoc "input_test_form_text"
                          (net.aserve:request-query req)
      :test #'equal))))
(princ
  (format nil
   "AJAX: ~A <a href=\"/search\"
               target=\"new\">click here</a>"
             test-input)
    net.html.generator:*html-stream*)))))
```

Handling file uploads is a bit tricky; I copied the code from the functions **fetch-multipart-sequence** and **process-upload-form** from the AllegroServe and WebActions documentations, adding new functionality to **process-upload-form** for processing the text of uploaded files and creating RDF triples in our local data store.

The last code snippet from file webapp.lisp that we will look at is the code for showing search results on the browser page:

```
(def-clp-function search_results (req ent args body)
  (let ((session (websession-from-req req))
        (test-input (cdr (assoc "input_test_form_text"
                                (net.aserve:request-query req)
             :test #'equal))))
    (push test-input (websession-variable session "history"))
    (net.html.generator:html
     (:princ
      (format nil "Search results: ~A" test-input)))
    (if test-input
        (dolist (doc-id (doc-search test-input))
          (princ
            (format nil
                 "<pre>~A</pre>~%" (get-doc-info doc-id))
                 net.html.generator:*html-stream*)))))
```

Figure 17.3 shows the login page.

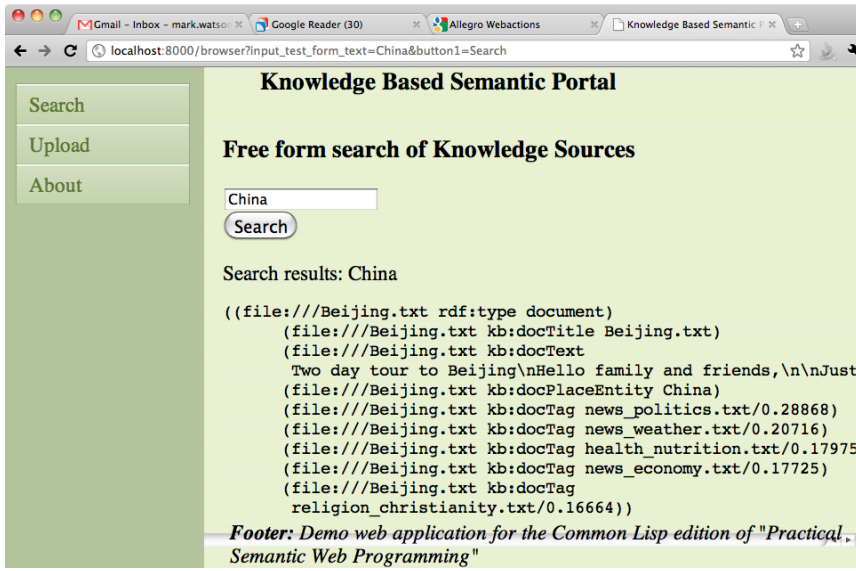

Figure 17.3.: Example Application Search Page

17.4. Web Application Wrap Up

There has been a lot of buzz in the industry about high level frameworks like Ruby on Rails and Django but relatively about Common Lisp frameworks like Portable AllegroServe and WebActions.[5] This is probably, quite frankly, because Common Lisp is not nearly as widely used as Ruby and Python. I am not even mentioning the rich ecology of Java server side frameworks - there are too many to mention.

That said, if you enjoy developing in Common Lisp, I hope that the brief introduction and example code in this chapter and in Chapter 16 will get you started using Lisp for the full stack of web application development.

I urge you to experiment with RDF for storing data. There is a lot of ”buzz” right now for so-called NoSQL databases like MongoDB, CouchDB, and Cassandra. RDF data stores predate these newer NoSQL data stores and offer many implementations to choose from.[6] Google, Yahoo, and other major web players are starting to add metadata to HTML web pages. When this metadata is in RDFa format it can be easily stored for reuse in an RDF data store. Other types of tags like microformats can also be converted to RDF.

[5] As well as other high quality frameworks like Edi Weitz’s Hunchentoot application server.
[6] For example, AllegroGraph, Sesame, Redland, Jena, 4store, the Talis Platform, and Virtuoso.

Index